

图灵程序设计丛书

WILEY

- 热门App的开发者现身说法
- Objective-C编程知识一应俱全
- 掌握移动互联时代最热门的开发语言

好学的Objective-C



Objective-C Developer Reference

[美] Jiva DeVoe 著
林本杰 译



人民邮电出版社
POSTS & TELECOM PRESS

Jiva DeVoe

拥有25年的软件开发经验，是专门开发iPhone和Mac OS X 应用的Random Ideas软件公司的创始人，已有多个iPhone应用成为苹果广告中的推荐应用。此外，他还是*Cocoa Touch for iPhone OS 3 Developer Reference*的作者。他的博客地址为www.random-ideas.net。

 图灵程序设计丛书

好学的 Objective-C

 Objective-C Developer Reference

[美] Jiva DeVoe 著
林本杰 译

人 民 邮 电 出 版 社
北 京

图书在版编目（C I P）数据

好学的Objective-C / (美) 德沃 (DeVoe,J.) 著 ;
林本杰译. -- 北京 : 人民邮电出版社, 2012.3
（图灵程序设计丛书）
书名原文: Objective-C Developer Reference
ISBN 978-7-115-27358-1

Ⅰ. ①好… Ⅱ. ①德… ②林… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2012)第004948号

内 容 提 要

本书共分为4部分。第一部分介绍了Objective-C的基础知识，包括Objective-C的基本语法、对象、内存管理等；第二部分深入挖掘Objective-C提供的一些功能，包括如何使用代码块，使用键值编码和键值观察，使用协议，扩展现有类的功能，编写宏以及处理错误和异常；第三部分介绍了Foundation框架及其相关知识；第四部分介绍了一些高级主题，包括多线程处理、Objective-C设计模式、利用NSCoder读写数据以及在其他平台上使用Objective-C等内容。

本书适合对Objective-C程序设计感兴趣的人阅读。

图灵程序设计丛书
好学的Objective-C

◆ 著 [美] Jiva DeVoe
译 林本杰
责任编辑 王军花
执行编辑 李 静
◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷
◆ 开本：800×1000 1/16
印张：16.25
字数：384千字 2012年3月第1版
印数：1－3 500册 2012年3月北京第1次印刷
著作权合同登记号 图字：01-2011-2966号
ISBN 978-7-115-27358-1

定价：55.00元
读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

Original edition, entitled *Objective-C Developer Reference*, by Jiva Devoe, ISBN 978-0-470-47922-3 , published by John Wiley & Sons, Inc.

Copyright ©2011 by John Wiley & Sons, Inc.，All rights reserved. This translation published under License.

Simplified Chinese translation edition published by POSTS & TELECOM PRESS Copyright ©2012.

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

本书简体中文版由John Wiley & Sons, Inc.授权人民邮电出版社独家出版。

本书封底贴有John Wiley & Sons, Inc.激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

致　　谢

写这本书很有收获并富有挑战，在这里我想感谢一些人，没有他们的帮助和支持我不可能完成本书。

首先，我想感谢Cynical Peak Software的Brad Miller，他是该公司最好的技术编辑之一。他对细节的关注以及在帮我改正错误方面的孜孜不倦的坚持真令人难以置信。谢谢你的努力。你太棒了。

我想感谢Wiley的Aaron Black的帮助和支持，尽管过程中有一些阻碍。非常感谢你的赞助和帮助。

感谢我的父亲Robert A. DeVoe，是你让我感受到技术的神奇并鼓励我在计算机方面追求梦想。

我儿子Alex花了很多时间帮我处理本书的格式，谢谢你。你的努力和帮助是本书完成的关键。

最后，也是最重要的，我要感谢我的妻子，感谢她不仅仅因为这个项目，而是因为她对我所有工作的一贯支持。没有她，本书就不会完成。你在我情绪低落和疲惫的时候鼓励我，并激励我追求新的成就和目标。真不知道该怎么感谢你。

序　　言

Objective-C在IT行业可谓受到了不公平的对待。尽管它很强大，并且是动态的面向对象语言，但却没有像C++、Java等语言一样得到足够认可。

在为iPhone OS 3写Cocoa Touch时，我意识到了需要写一本配套的书，以帮助新手们在接触Cocoa和Cocoa Touch等高层框架之前克服学习Objective-C的障碍。

所以当有人请我写一本专门介绍Objective-C语言的书时，我欣然接受了。

最后，我感觉到可以通过这本书向Mac、iPhone和iPad开发新手们介绍基础知识，因此万分激动。我期待这本书可以催化Objective-C在更多不同平台上发展。Objective-C完全有理由在Unix、Windows等平台上使用。

读者只需具备有限的计算机知识。我会从最基础的知识开始阐述，但是你至少需要懂得一些操作计算机的基础知识。

如果你已经熟悉了其他一些编程语言，这也不会有任何负面影响。我介绍的一些东西对你而言可能是一种回顾，不要担心，你会学到很多关于Objective-C的细节。

如果你接触过Objective-C，希望你可以在本书中发现一些有价值的新信息。我会努力将这些知识设计得便于你查找。这样一来，你无需逐页浏览，就能跳到某一部分并了解如何完成你想完成的任务。

对于本书中使用的一些约定，我尽量确保一致，同时尽量遵照苹果的约定。唯一一个比较明显的例外就是使用“方法”来表示实例和类的函数。苹果通常会倾向于使用“消息”。某种程度上这是缘于Objective-C受到Smalltalk的影响。

关于键盘快捷方式，我选用“Command键”这一术语来表示多数苹果键盘上空格键左侧的键。大家可能知道它也叫苹果键，因为就在几年前它上面会印有一个苹果标志。此外Command键旁边的键称为Option键，Option键旁边的就是Control键。 这些是和苹果文档的约定保持一致的。

关于存储对象的变量，我通常会把它们称作“实例变量”。有些书会习惯用该术语或者其缩写“ivar”来指代作为类的一部分的变量。对此，我喜欢使用“成员变量”。在我看来，成员变量可以是实例变量，但不是所有的实例变量都是成员变量。

在文中提及方法时，我会遵照苹果引用它们的约定：使用方法名，但不包括参数。比如以下方法：

```
-(void)someMethodUsingParam1:(NSString *)param1 andParam2:(NSString *)param2;
```

就会被写做：`-someMethodUsingParam1:andParam2`。如果它是一个类方法，打头的连字符就会被替换成一个+号，就像你在写类定义中的方法一样。

关于示例代码，在需要构建完整项目的章节，通常我会尽可能提供代码的完整列表。在没有提供的情况下，你可以从本书网站上下载包含图片资源和其他相关支持文件的项目。有部分章节可能无法创建一个完整的项目来展示相关技术。在这种情况下，代码列表可能只是一些片段，你可用作自定义代码的基础。由于这些代码片段无法构成功能完整的项目，在网站上也就没有提供示例项目。

我希望你在阅读本书时会有一种和我写作时一样的愉悦体验。在我看来，一本好的技术书的标志就是它不会被束之高阁。它会被好好地放在书桌上或者书桌旁，因为经常需要翻阅它。我希望这本书在你的手中也会有这样的地位，并且希望它书角翘起、封面破损，每页都留有潦草的笔迹，但仍然能在未来几年对你有所帮助。

Jiva DeVoe
book@random-ideas.net

目　　录

第二部分 更多特性

第四部分 高级主题

Part 1

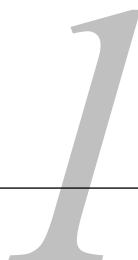
第一部分

Objective-C 简介

本部分内容

第 1 章

Objective-C 简介



本章概要

- 学习 Objective-C 历史
- 了解编写 Objective-C 代码的 Xcode
- 配置开发环境

这一年是 1986 年，是哈雷彗星 75 年来最接近太阳的一年。英国和法国宣布建造英法海底隧道的计划。宝丽来盛行并刚刚迫使柯达退出了快速相机业务。人们使用 C 语言差不多 15 年，而 C++还是该领域的新军并鲜为人知。Smalltalk 语言在技术界打了个翻身仗，人们开始对一种称作面向对象编程（缩写是 OOP）的新概念感到兴奋。

Tom Love 和 Brad Cox 这两名开发人员在 ITT 公司的编程技术中心接触了 Smalltalk。Cox 想，要是在 C 语言中加入面向对象功能，只用 C 就可以进行面向对象编程，那定会很有意思。实际上，他将这种扩展命名为 COOPC，表示是用 C 实现的面向对象编程。最终，两个人成立了一家公司来商业化这些扩展并将其作为一种语言向开发人员推销。这一新语言也更名为 Objective-C。若干年后，Steve Jobs 领导的一家名为 NeXT 的小型创业公司，获准使用并标准化了 Objective-C，以作为将要开发的 NeXTstep 操作系统的主要语言。NeXT 计算机公司最终被苹果收购，NeXTstep 操作系统最终发展成为 Mac OS X。

很少有人会想到 Objective-C 历史悠久，并且它实际上影响了很多其他的编程技术。比如，Java 编程语言和 Objective-C 就有很多共同点。原因就是在 Objective-C 的早期，NeXT 和 Sun Microsystems 合作开发 OpenStep 平台，他们用来开发这种技术的语言就是 Objective-C。当 NeXT 计算机的表现没有达到他们预期的要求时，该公司走向了失败，Sun 决定开发自己的语言和跨平台开发包——Java。Java 工程师们都是谙熟 Objective-C 的，因为 Objective-C 是他们在使用 Java 之前首选的语言。后来他们就将 Objective-C 的一些较好的功能引入到了他们所开发的语言中。

Objective-C 现已成为了 Mac OS X 和 iPhone OS 上首选的开发语言。它已经发展成为了一种优雅的解决方案，在纯静态语言和纯动态语言之间实现了平衡。它是少有的几种通常进行编译的语言，不仅能从类似 C 和 C++的编译时语法检查受益，还能从负责处理动态对象类型的动态运行时受益。

除了 Mac OS X 和 iPhone OS，Objective-C 在其他平台上也发展了一批追随者，可以在 Linux、Windows 和其他支持 GNU 编译器的平台上开发应用。在 iPhone OS 上的使用增加了该语言的知

名度并吸引了很多新的程序员。可以说 Objective-C 如今正在经历一次复兴——成千上万的开发者正涌向该语言，使其成为了最热门的技术之一。

本书将介绍 Objective-C，并展示为什么我觉得它也是世界上一流的编程语言。我觉得一个好的程序员需要三种语言。第一种是工作流程自动化语言。通常这是一种脚本语言，可用于自动化工作空间并构建一个用于优化工作流的临时工具。第二种是编辑器宏语言。作为程序员，我们会花 99%的时间用于将文本打造成软件。有一个可以帮助你控制编辑器的重要工具。最后一种是用于构建系统和应用的语言，可以用于部署要求高性能和高稳定性的应用。通常这些语言都是编译型的，这样你就可以从所选平台中获得最佳性能。但是，这些语言最重要的特点就是可以最大限度地利用系统库。

希望你看完这本书后，Objective-C 会成为你的应用开发语言。对于实现各种任务，和其他语言相比，该语言绝对略胜一筹。

1.1　使用 Xcode 进行开发

本书假定你使用 Xcode 开发环境进行编码。Xcode 是一个加入苹果开发者计划就可以免费获得的优秀的 IDE。它默认支持 C、Objective-C、C++、Java 以及其他几种语言，不过本书中我们只用它来编写 Objective-C 程序。

1.1.1　新建项目

首先启动 Xcode，你可以选择打开最近打开的项目或者创建一个新项目。为了便于讨论，选择新建一个项目，这样大家就可以跟着练习。之后会弹出一个如图 1-1 所示的“新建项目”对话框。

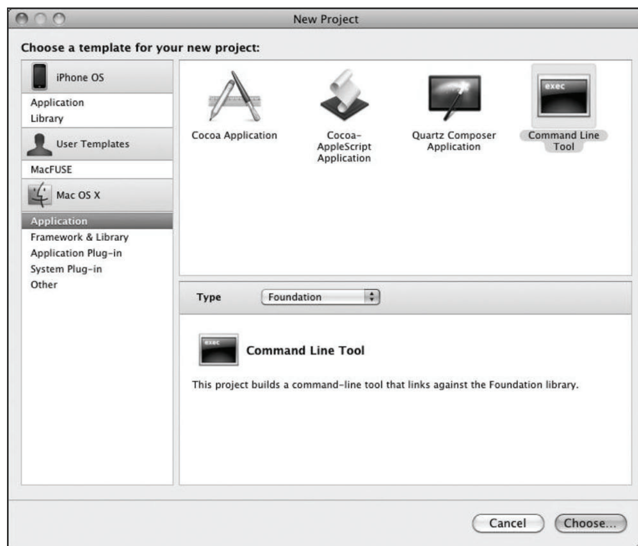


图 1-1　“新建项目”对话框

在该对话框中，你可以选择创建各种不同类型的项目。从命令行应用到桌面图形应用，你可以找到几乎所有的模板。此外，你如果安装了 iPhone SDK，就有多种不同的 iPhone 和 iPad 应用的模板。由于目前我们主要关注的是 Objective-C，选择其中最简单的一种项目就好了。

(1) 从 Mac OS X 组中选择 Application 后选择 Command Line Tool。

(2) 在 Type 的下拉列表中选择 Foundation。

(3) 单击 Choose 按钮，选择一个保存新项目的位置后单击 Finish。

在下面几节中，我们会简要介绍 Xcode 开发环境，这样你就可以慢慢熟悉它。我们就从图 1-2 所示的 Xcode 窗口开始。

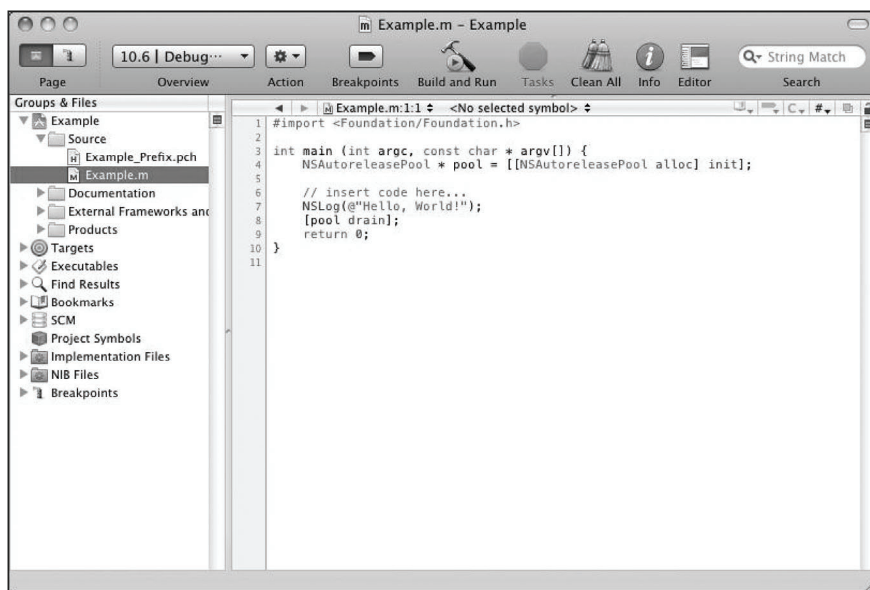


图 1-2　主 Xcode 窗口

主 Xcode 窗口包括两个面板：左边的面板包含了项目中的所有文件。选择其中的一个文件就会在位于窗口右侧的编辑面板中显示。在 Xcode 中，可以将项目文件移到项目中的目录中进行分组。大多数情况下，这些组仅仅在开发期间有用，对最后完成的项目影响很小，甚至没有任何影响。

除了源码文件外，还显示了链接项目所需的框架。

在项目文件下方是一系列的智能组。这包括了项目将要生成的目标、搜索结果和断点等。

目标组包括了项目编译生成的各种目标。改变组内对象的设置，就可以重写项目范围的编译设置。在这里还可以为项目添加和编辑一些自定义编译步骤。该组的编译设置如图 1-3 所示。

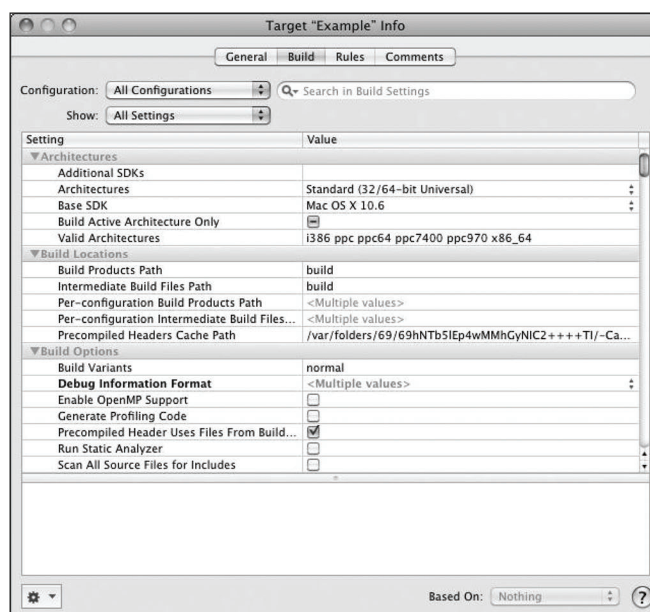


图 1-3 编译设置

1.1.2 项目文件

在这个简单的项目中，源码文件包含在源码文件组内。目前你可以看到只有一个源码文件，文件名和项目名是一致的。文件的扩展名是.m。单击该文件应该会在 Xcode 的编辑器面板中显示它，如图 1-4 所示。

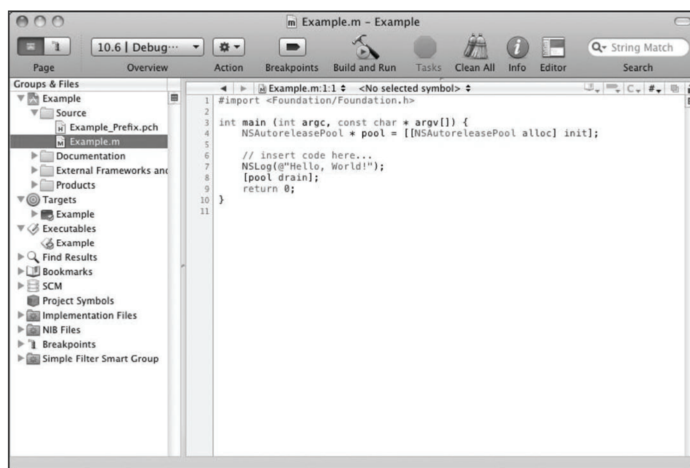


图 1-4 Xcode 中的编辑器面板



说明

虽然我们只提到一个源码文件，但这里其实还有一个扩展名为.pch 的文件。这是预编译头文件，不需要我们编辑或处理，它是编译器自动生成的。

不必担心看不懂文件中的源代码，下一章会介绍 Objective-C 的基本语法。现在，关键是要理解 Xcode 以及它的工作原理。



说明

如果没有显示源文件，可能就需要拖动 Xcode 窗口的底部的分割条来显示源码编辑器。

默认项目中包括的其他文件有：文档组（Documentation）中的一个程序文档文件、外部框架和库组（External Frameworks and Libraries）中的项目相关的框架，以及位于产品组（Products）中的可执行文件。现在的可执行文件是红色的，这是因为项目还没有编译出可执行文件。如果单击编译和运行（Build and Run）按钮，就会编译可执行文件、运行它并在控制台窗口中显示输出结果。好好熟悉一下控制台窗口，因为接下来的几章会经常使用它来检查所编写程序的输出结果。

1.1.3　添加源码文件

在项目中新建一个源码文件，你可以选择文件组织面板中的源码文件组，然后单击 File ➤ New File 菜单，之后就会弹出一个如图 1-5 所示的“新建文件”对话框。

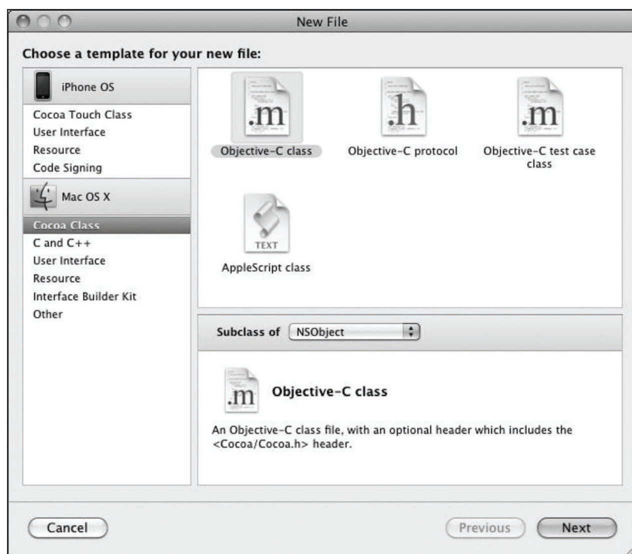


图 1-5　“新建文件”对话框

本书中需要添加文件时，大多数情况下会使用 Cocoa 类选项中的 Objective-C 类模板，所以请熟悉该窗口。

在有些情况下，一个项目中会有多个目标，这样就可能有编译成不同目标的不同源码文件。显式地在当前选择的目标中包含或者不包含一个经过编译的文件，可以单击该文件，然后找到源码面板中的详情视图。小小的目标栏中的复选框处于选中状态，这表示该文件被配置为针对当前目标编译。取消选中后，就不会被编译，如图 1-6 所示。

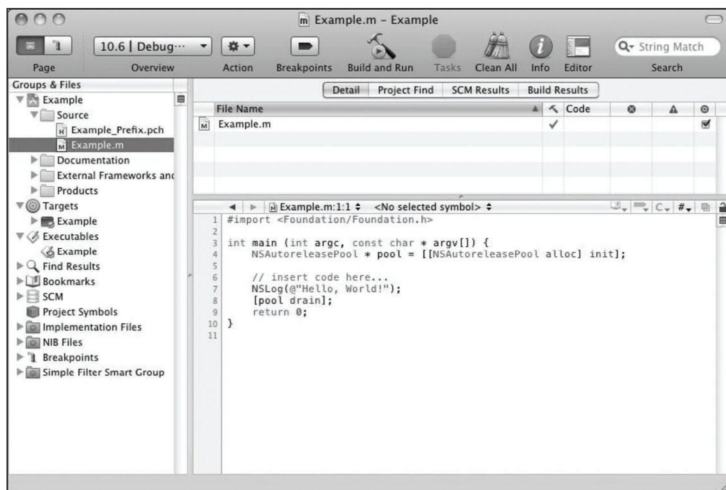


图 1-6　显示“目标”复选框



说明

如果你看不清详细显示面板，可以选择 View ➤ Zoom Editor Out 菜单来显示。

1.1.4　主Xcode窗口

现在有些熟悉 Xcode 基本的文件管理了吧？接下来我们再看一看主 Xcode 窗口，这是你使用 Xcode 时完成大部分工作的地方。主 Xcode 窗口如图 1-7 所示。

查看该窗口时，你会看到窗口的左侧是文件浏览器（File Browser）面板，右侧是详细显示面板，或者也可以称为编辑器面板，本书剩余部分将采用这种叫法。选择文件浏览器面板中的文件就可以在右侧的编辑器面板中显示它。此外，编辑器面板有几种不同的模式。详细模式显示了文件浏览器中所选择文件的简要说明。项目查找（Project Find）模式显示一个查找面板，这样就可以在项目的所有文件中查找任意字符串。如果在 Xcode 中开启了 All-In-One 布局，那么编辑器面板中就多了一种编译结果模式，这样就可以看到上次编译的结果了。

现在看看文本编辑器窗口上方的小条。这里显示了当前编辑的文件的一些有趣信息。

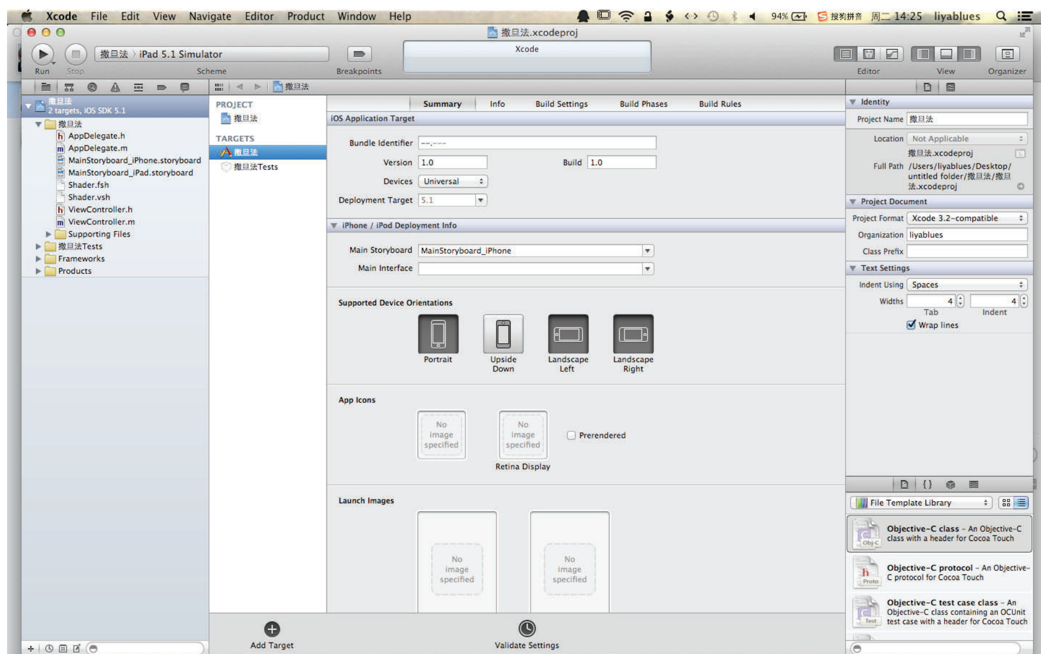


图 1-7　主 Xcode 窗口

在顶部小条中看到的第一项就是当前编辑文件的文件名和行数。这实际是一个下拉列表，你可以从最近打开的文件列表中单击选择。旁边是另一个下拉列表，显示了当前编辑的文件中方法的函数声明。如果选择其中一个方法，编辑器就会自动跳转到当前文件中该声明的位置。

图 1-8 显示了一个下拉列表处于展开状态的文件。

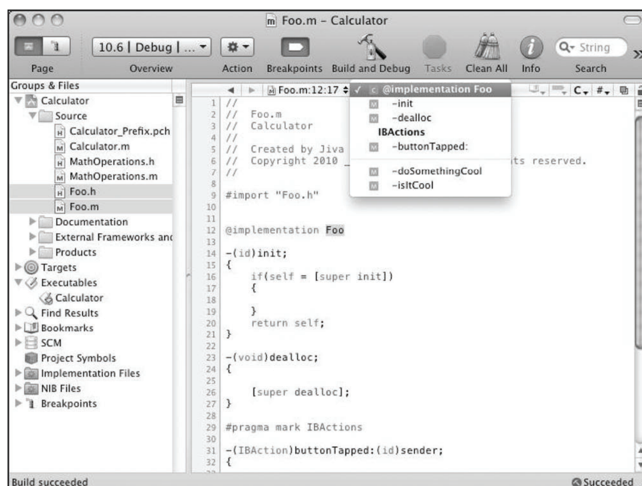


图 1-8　文件导航下拉列表



说明

你可以通过在代码中使用 pragma 指令向该下拉列表添加任意的文本。对于该文件中，你可以看到已经有一个界面构建器动作方法的 pragma 指令。

在文本编辑器和文件浏览器的上方是主工具栏。你可以通过不同的按钮进行配置，但通常采用默认值即可。通过这些按钮可以启动编译、停止当前编译等。

最后，在 Xcode 窗口的右下角可以看到上次编译的结果。如果编译成功就可以看到 Succeeded。如果失败了，可以看到上次编译过程中产生的警告和错误数，黄色图标表示警告，红色图标表示错误。如果最近运行的是静态代码分析器，生成的任何警报都用蓝色图标表示。

1.2 理解编译过程

在开始详细介绍 Objective-C 的语言特性之前，你必须全面理解编译过程。编译过程是计算机将敲入的代码转换成可以真正执行的指令的过程。最终，计算机实际只能执行用它们的“母语”（机器语言）编写的指令。这样的语言通常极其繁冗，人类很难理解和使用。因此，在编码时我们使用的是 Objective-C 这样的高级语言。我们在文本编辑器中编写代码，保存文件到硬盘，然后针对文本文件运行编译器。编译器接收文本并将其转化成计算机可以执行的指令。

这其中的大部分都是基础知识。你要是已经很熟悉编程（可能用另一种语言），就可能会知道这些，或许可以跳过接下来的几节。但如果你是编程新手，这些知识就很有用了。尽管实际上编译过程是很短的，但理解这个过程还是很有意义的。

理解编译过程的第一步就是编写代码。

1.2.1 编码

前面提到，作为程序员你会花 99%的时间将文本打造成软件。在所有的软件开发中，程序员在编译前都需要将计算机指令敲入到一个文本编辑器中。这通常就称为“编码”。

实际上这无非就是敲出指令并将指令保存为文本文件。我们通常将这些文件及其包含的指令称为源代码。

很多编程语言都包含接口和实现的概念。接口通常是一个模块公开给另一个模块的方法和属性。接口实现就是为了兑现对接口中另一个模块的承诺，而让计算机实际执行的指令。换句话说，接口就是系统的一个模块对其他模块承诺它可以干什么，而实现就是兑现承诺所需的计算机指令。

大部分的编程语言根据对接口和实现的处理方式可以分成两种类型。第一类语言不分离接口和实现，它们只使用一个文件在同一个位置声明接口和实现。第二类语言分离接口和实现，使用两个不同的文本文件单独表示接口和实现。Objective-C 就属于后者。

Objective-C 是面向对象的编程语言。这就意味着开发者需要将程序的不同组件划分成不同的对象和类。类是数据和操作数据的方法的集合，对象就是类的一个单独的实例。Objective-C 类包含一个接口和一个实现。

用于存储 Objective-C 源代码的文本文件名通常有一个.m（实现文件）或者.h（接口文件）扩展名。比如，在创建一个名为 `MyClass` 的类时，会为接口和实现分别创建文件名为 MyClass.h 和 MyClass.m 的文本文件，分别用来保存接口和实现对应的计算机指令。代码清单 1-1 和代码清单 1-2 显示了上述两个文件。目前，不必急于理解它们的内容。这里列出来就是让大家感觉一下所谓的实现是什么模样。

代码清单 1-1　接口文件

```
@interface MyClass : NSObject
{
    int foo;
}
@property (nonatomic) int foo;
-(void)someMethod;
@end
```

代码清单 1-2　实现文件

```
#include "MyModule.h"

@implementation MyClass
@synthesize foo;

-(void)someMethod
{
    NSLog(@"some method got called");
}
@end
```

再次提醒一下，目前不要急于去理解这些代码，但我要借此机会介绍上述代码清单中一些有趣的特征。

首先，在接口文件中，`@interface` 指令清晰地表明这是一个接口。接口以`@interface` 开始并以`@end` 结束。这里可以分为几个部分。

第一部分位于接口头部，通过大括号（`{}`）分隔。在这里声明成员变量。成员变量存储和该模块相关的数据。

第二部分用于声明属性和方法。属性记录并控制对对象状态和数据的访问。方法是用于操作数据的计算机指令。上述代码就是一个典型的面向对象模块。第 3 章将介绍更多关于面向对象编程、类、成员变量、属性和方法的相关知识。



说明

Objective-C 保留了 C 语言的基本特性。因此，在 Objective-C 中用纯 C 创建一个模块是不可能的。在这种情况下，模块中的接口文件和实现文件的扩展名分别为.h 和.c。此外，如果你在命名实现文件时使用.cc 作为扩展名，Xcode 会按照 C++来编译，使用. mm 作为实现文件的扩展名时，会将其当成 Objective-C++（这就是 Objective-C/C++混编）来编译。在本书中，我们将学习 Objective-C，因此就不会进行此类操作。

你可以使用任何可以保存纯文本文件的文本编辑器进行 Objective-C 编码。不过苹果在 Mac OS X 上提供了一个卓越的 Xcode 集成开发环境，适用于 Mac OS X、iPhone 和 iPad 应用开发。这真是一个绝好的工具，同时也是本书选用的工具。当然如果在其他平台上进行开发，就需要寻找一个适合所选平台的文本编辑器。

1.2.2 源码、编译代码和可执行文件

在创建源码后，需要计算机将源码编译成它可以执行的指令。这个过程就称作编译源码。

编译源码通常包括下面几个步骤。

第一步称为*预编译*。可以将预编译想象成计算机为编译代码所进行的准备。在这一步中，编译器会移除一些注释等不会变为可执行程序的代码。它同时也会展开部分代码并重新排列某些指令，以使得编译的第二步更高效。编译第一步的结果就是一个源代码的中间状态。你通常不会看到或处理代码的这种中间状态，但利用编译器选项可以在你想看输出结果时让编译停止在该阶段。一些进行高级开发的程序员有时会使用这些编译选项来查看中间文件和编译器具体的工作过程。在通常的应用开发中可能永远也用不着这样做。

预编译后，第二步就是编译器将源代码真正转变成目标文件。目标文件的扩展名是.o。编译源码可能会花很长时间，因为要编译很多不同的模块和不同的源码文件。在某些模块自身以及依赖的模块都没有改变的情况下，不重新编译这些模块可以大大缩短编译时间。因此，目标文件通常会存储在编译目录中。在上次编译后源码文件没有改变的情况下，编译器就会跳过该源码文件的编译并复用上次编译产生的目标文件。通常，在日常使用中也不需要查看这些文件。它们仅供编译器使用。编译的最后一步称作*链接*。链接就是将上一步生成的目标文件连接起来以形成可执行程序。除了目标文件外，库和框架也会被链接到可执行程序中。链接过程的结果就是实际的应用可执行文件。在命令行应用中，链接的结果就是一个可以在命令行运行的二进制文件。而在桌面应用中，结果通常是一个应用包，也就是硬盘上的一个包含可执行程序和图片、声音等运行应用所需资源的目录。下一节我们就会介绍应用包。

在 Xcode 中，单击 Build 和 Run 按钮就可以编译应用。如果在示例项目中这样操作，就会在控制台启动应用并显示它的输出，这样 Xcode 就会切换到调试模式，这时你也可以调试应用。调试是一个高级 IDE 主题。关于如何使用 Xcode 进行调试可以参考 Xcode 文档。

编译过程的最终结果就是可执行文件。如果编译器在任意时刻发现源码中的错误（实际上这是很常见的），就会停止处理该文件并显示错误，这样你就可以改正错误，可惜的是编译器相当挑剔。在通过文字推断含义方面，电脑永远逊色于人类。结果就是编译器不会猜测你想敲入什么代码，而是直接放弃并显示错误。这些错误简单到漏写了分号、漏写了空格、不正确的大写以及许多琐事，请准备好——选择了程序员生涯，就要每天面对成百上千个此类错误。即使最优秀的程序员也很少能写出第一次就可以完美编译的代码。

1.2.3 查看应用包

在上一节中，我提到了应用包这个词——即使你是一个有经验的程序员，对你来说这也可能

是个陌生的词。你可能想知道这到底是什么。与其说应用包是一种 Objective-C 结构，不如说它是一种操作系统结构。Objective-C 语言本身并不需要或者生成应用包。尽管如此，应用包对于使用 Objective-C 的几乎所有平台上的编程来说仍是一个重要的、必不可少的概念。这是作为 Objective-C 程序员需要理解的一个重要概念。

应用包只不过是硬盘上包含一组文件的目录。在 Mac OS X 中，应用包可以用来集合应用运行所需的所有文件。这包括可执行程序、图片、音频文件和用户界面资源等。此外，在为 Mac OS X 编译界面应用时，用户界面定义通常是在名为 Interface Builder 的应用中完成的。该应用也会生成名为 NIB 文件的包。NIB 文件通常有.nib 后缀并存储在应用包内部。



说明

NIB 的全称是 NeXTstep Interface Builder，这是 NeXT 时代的遗留物。最近，它的格式从二进制变成了 XML，相应的文件名后缀也从.nib 变成.xib。在编译过程中，XIB 文件仍然会被编译成 NIB 文件，所以在应用包中，你会看到.nib 而不是.xib。

代码清单 1-3 显示了一个典型的应用包目录结构。在本例中，这是 Xcode 应用包的一部分。

代码清单 1-3　应用包目录结构

```
Xcode.app
`-- Contents
|-- CodeResources -> _CodeSignature/CodeResources
|-- Info.plist
|-- Library
|    |-- QuickLook
|    |    `-- SourceCode.qlgenerator
|    |        `-- Contents
|    |            |-- CodeResources -> _CodeSignature/CodeResources
|    |            |-- Info.plist
|    |            |-- MacOS
|    |            |    `-- SourceCode
|    |            |-- _CodeSignature
|    |            |    `-- CodeResources
|    |            `-- version.plist
|    `-- Spotlight
|        |-- SourceCode.mdimporter
|        |    `-- Contents
|        |        |-- CodeResources -> _CodeSignature/CodeResources
|        |        |-- Info.plist
|        |        |-- MacOS
|        |        |    `-- SourceCode
|        |        |-- _CodeSignature
|        |        |    `-- CodeResources
|        |        `-- version.plist
|        `-- uuid.mdimporter
|            `-- Contents
|                |-- CodeResources -> _CodeSignature/CodeResources
```

```
|               |-- Info.plist
|               |-- MacOS
|               |   `-- uuid
|               |-- Resources
|               |   |-- English.lproj
|               |   |   |-- InfoPlist.strings
|               |   |   `-- schema.strings
|               |   |-- Japanese.lproj
|               |   |   |-- InfoPlist.strings
|               |   |   `-- schema.strings
|               |   `-- schema.xml
|               |-- _CodeSignature
|               |   `-- CodeResources
|               `-- version.plist
|-- MacOS
|   `-- Xcode
|-- PkgInfo
|-- PlugIns
|   |-- BuildSettingsPanes.xcplugin
|   |   `-- Contents
|   |       |-- CodeResources -> _CodeSignature/CodeResources
|   |       |-- Info.plist
|   |       |-- MacOS
|   |       |   `-- BuildSettingsPanes
|   |       |-- Resources
|   |       |   |-- Built-in Build Settings Panes.pbsettingspanespec
|-- Resources
|   |-- AskUserForNewFileDialog
|   |-- CreateDiskImage.workflow
|   |   `-- Contents
|   |       `-- document.wflow
|   |-- DevCDVersion.plist
|   |-- Document-Cert.icns
|-- _CodeSignature
|   `-- CodeResources
`-- version.plist
```

需要注意的是 Contents/MacOS 目录中包含 Xcode 可执行程序。该可执行程序和命令行应用生成的可执行程序是一样的。不同的是该可执行程序被打包到了应用包内部并且包含从包加载资源的代码。应用中包含一些有意思的目录，包括 PlugIns 目录，其中同样也有包。Spotlight 目录包含以 SourceCode.mdimporter 和 uuid.mdimporter 目录形式存在的包。本书的第二部分将要介绍的 Foundation 框架包含一些读取应用包的方法，并且该框架支持访问这些内嵌包。

目前你需要知道的重要一点就是：创建 Mac OS X、iPhone 或者 iPad 的图形应用时，Xcode 会创建一个应用包。今后如果需要在应用中包含分隔开的资源组，你就需要自己创建应用包。Xcode 也支持这样的功能，不过目前你还不需要考虑这些。

1.2.4　编译设置

Xcode 中有两处可以设定项目的编译设置。第一处也是主要的一处就是项目信息窗口，可以

通过选择 Project ➤ Edit Project Settings 打开该窗口。在本节中，由于这些信息很有用，所以我会详细介绍这些窗口。由于这属于本书讨论的范畴，如果我不介绍你就可能无法理解其中的很多东西。我建议你略过本节并在学完第 2 章后再回过头来学习。如图 1-9 所示，项目设置窗口的第一个设置面板是项目的通用设定。其中的大部分都不需要改变，不过你可以利用 Configure Roots and SCM 按钮配置源码管理方式。

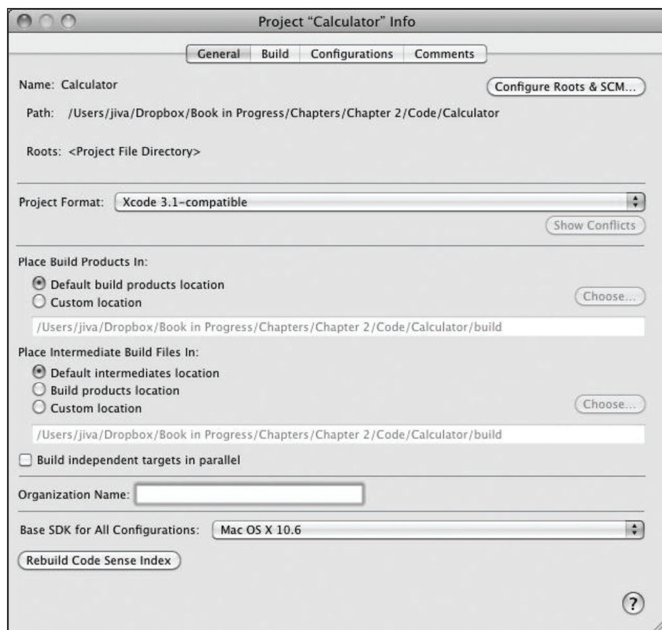


图 1-9　项目设置

Place Build Products In 设置用于配置编译后可执行文件在项目中的存放位置。再次指出，这也是一个通常不需要改变的设定，但它在需要确认可执行文件存放位置的时候就很有用。在该目录中可以找到可执行文件。

这之后的设置 Place Intermediate Build Files In 用于设置编译时源码的目标文件的存放位置。

接下来的 Build Independent Targets in Parallel 设置会影响编译器为不同的独立平台（比如 PPC 和 Intel）编译目标文件。如果勾选该选项，就会并行编译不同的目标，否则就会逐个编译。

使用 Xcode 时经常会问的一个问题就是“如何修改自动创建的位于源码文件头部版权声明后的公司名字？”接下来的 Organization Name 就可用于设置公司名称。在这里设定公司名后，如果向项目添加新文件，版权声明中就会将在此设定的公司名作为版权的所有方。应该将 Organization Name 设置成公司名或者人名。

接下来的 Base SDK for All Configurations 用于配置应用默认要编译和链接的 SDK。所使用的 SDK 规定了项目中可用的代码补充和框架。阅读本书以及练习项目时可以将基准 SDK 设定成当前 Mac OS 的版本。但是，如果你是在其他平台上进行开发，比如 iPhone OS 等，就要合理配

置该设置。在将应用编译成一个更早版本的 Mac OS 时也需要修改该设定。



说明

SDK（Software Development Kit，软件开发套件）是库、工具、文档及源码文件的集合，用于在特定的平台上编译应用。Xcode 自带了 Mac OS X 和 iOS 的 SDK。

该面板上的最后一个设置是 Rebuild Code Sense Index。你可能会遇到代码感应索引（Code Sense Index）被破坏的极为罕见的情况。在这种情况下，你可以使用该按钮重新创建代码感应索引。这种情况很少见，但如果需要这样做，就可以在这里实现。

项目的编译设置可以在两个地方进行。第一个就是项目级的，第二个就是单个目标文件级的。如果你想将项目级的设置作为项目编译的基准设置，那么目标文件级的设置就是针对特定目标而需要改变的项。比如，一个给定的应用的项目有调试目标和发布目标。在本例中，可以将两个目标的项目级设置设置成一样的，但也可以有一些不同，比如是否需要裁剪可执行文件等。Xcode 通过使用两个单独的窗体来包含项目范围的设置和目标范围的设置，从而支持这一操作。如果进行项目级设置，并且没有更改特定目标的设置，那么项目的设置结果也会影响到目标。如果对特定目标的设置进行更改，那么该更改就会重写项目级的设定。图 1-10 显示了这些设置。

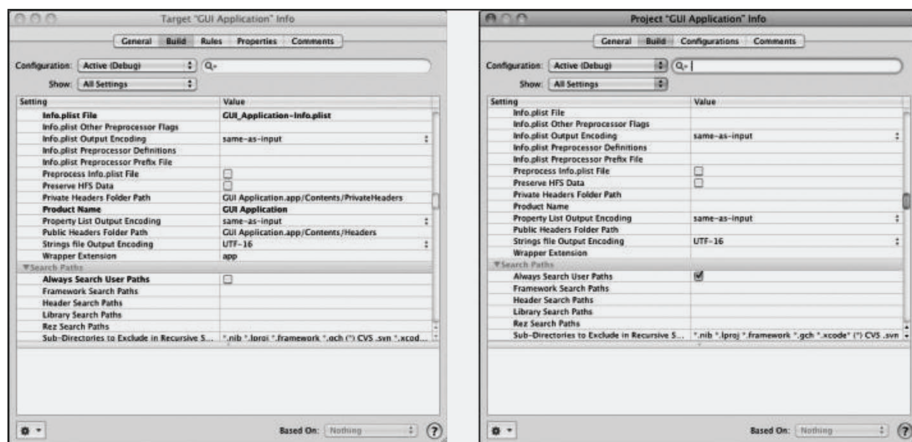


图 1-10　目标设置和项目设置

在该示例中，可以同时看到项目级设置和目标级设置。该示例应用配置了两个共享相同代码但编译不同可执行文件的不同目标。可以看到，对于项目级设置和目标级设置不同的项，其项目级设置会用黑体显示。如果选择删除项目级设置，黑体就消失了，取而代之的是从项目级设置获取到的默认值。

屏幕顶部的下拉菜单用来选择所要编辑的配置，比如是调试编译还是发布编译，以及用于过滤特定设置的列表。在项目层面上显示的配置选项是用 Configurations 选项卡设置的。图 1-11 显

示了一个配置了调试编译和发布编译的常见项目。

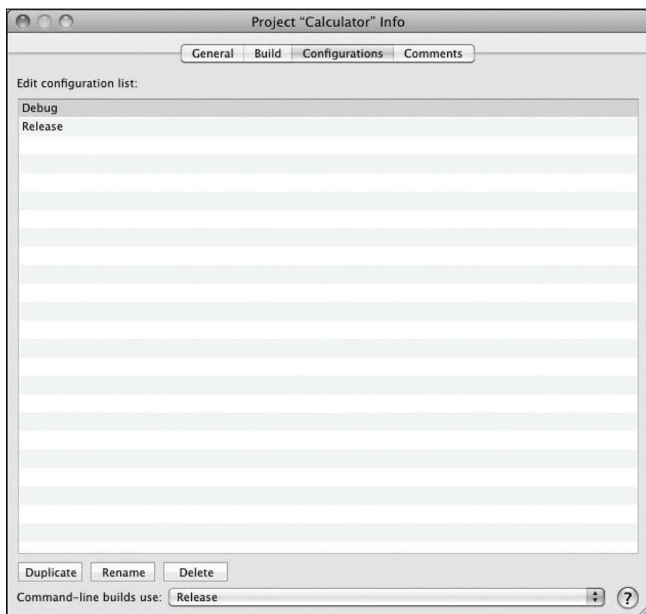


图 1-11　编译配置

你可以将配置看做相同目标的不同编译版本。除了调试和发布的编译版本之外，另一种常用情形就是你需要针对不同的架构进行编译，比如 PowerPC 和 Intel。

通过 xcodebuild 命令行工具编译项目也是可行的。这在编译自动化的时候就很好用。在这种情况下，窗口下方的选项 Command-line builds use 就可以控制在未指定配置的情况下所使用的默认配置。

对于本书的大部分内容来说，大多数设置都不需要改变。也就是说，如果你想了解各个设置的用处，就可以滚动设置页面，单击其中一个并查看屏幕下方显示的描述各个选项的信息。目标编译设置中的第三个标签页是 Rules，可以用以配置项目中不同类型文件的默认编译行为。比如，如果某种特定类型的文件在编译过程中需要特殊处理，就可以单击添加按钮添加该文件。然后配置任何你所需要的自定义行为类型。此外，你如果需要改变任何一种内置文件的默认行为，也可以通过下拉列表选项来改变。

大多数情况下，你不需要对设置进行任何改变。在编译带图形界面的 Cocoa 应用时，在目标编译设置中一个额外的选项卡就是 Properties。该选项卡可用于配置项目的可执行文件名以及主包标识符等。通常，你会把标识符配置成和你的公司名相匹配，而不是使用默认值。此外，如果想要通过双击打开应用所保存的文件，可以为这些文件注册并配置一个文档类型。在编译应用时，Mac OS X 会检测到你所指定的和应用相关联的文件类型，并且会列出你的应用，表示可以打开这类文件。图 1-12 就显示了这种行为。

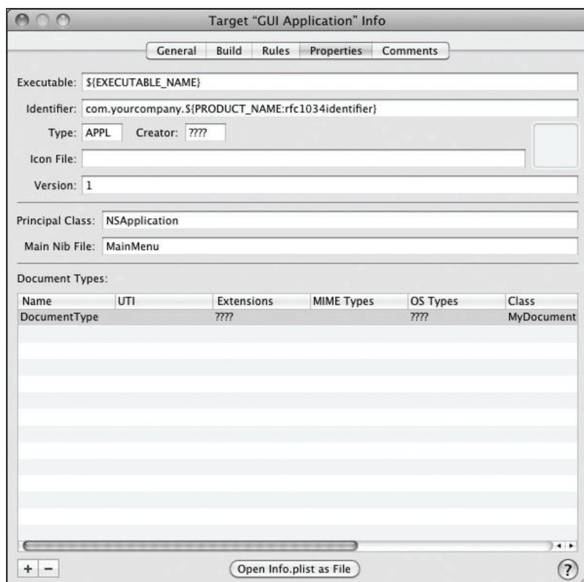


图 1-12　应用包属性



说明

对于命令行应用，就看不到 Properties 选项卡。Properties 选项卡实际上配置的是应用包的信息，而这在命令行应用中是不存在的。

如果是图形应用，还可以设置应用图标。可以通过在应用包中包含一个图标文件并在 Icon File 设置中指定文件名实现。

对于本书中的大部分开发，你不需要改变其中的任何一个选项，但是知道这些对以后很重要。



说明

本书印刷时还在开发中的 Xcode 4 对编译设置位置改变了一些，但设置本身几乎是一样的。关于该功能的最新信息请参考 Xcode 4 文档。

1.3　使用 Xcode 静态分析器

近几年来在 Xcode 开发环境中，编译器技术的最大的改进之一就是引入了 Clang 静态分析器。Clang 静态分析器是一个通过分析源代码来发现常见错误的工具。尽管编译器擅长发现一些错误，但通常会考虑速度而放弃了对一些较难发现的条件的检查。这样一些原本可以发现的错误和一些

在代码检查中可以发现的错误，经常会未被发现而成为应用中的 bug。未成功释放已分配的内存、死循环、使用未初始化的变量等都是这类错误的例子。普通编译器无法检查到这其中的大部分错误。Clang 静态分析器就是为了满足这种需求而专门设计的。

要想在源代码上运行 Clang 静态分析器，需要选择 Build ➤ Build and Analyze。这首先会利用编译器编译代码，然后运行静态分析器。

静态分析器显示检查到的错误的方式和普通的编译警报相似。但是在单击源代码中的一个错误时，你可以获得额外的上下文信息，这些信息采用图形代码流向箭头的形式。这些箭头会指示分析器预测的代码运行时使用的代码路径。这些信息可以帮助你更详尽地了解分析器在发现代码错误时考虑到的具体情况。

看看分析器如何处理常见的编码错误。再次提醒，这里讨论的大部分内容都是后面的章节会详细介绍的高级主题。目前，可以跳过本节，以后在自己的代码中发现这些错误时再回过头来看看这部分。

代码清单 1-4 显示了一个带有常见错误的示例程序。在本例中，已分配的内存没有被释放。

代码清单 1-4　带有内存泄漏的程序

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSDate *date = [[NSDate alloc] init];
    NSLog(@"The time is: %@", date);
    [pool drain];
    return 0;
}
```

在本例中，`NSDate` 对象创建后没有被释放。这段代码的静态分析器输出如图 1-13 所示。

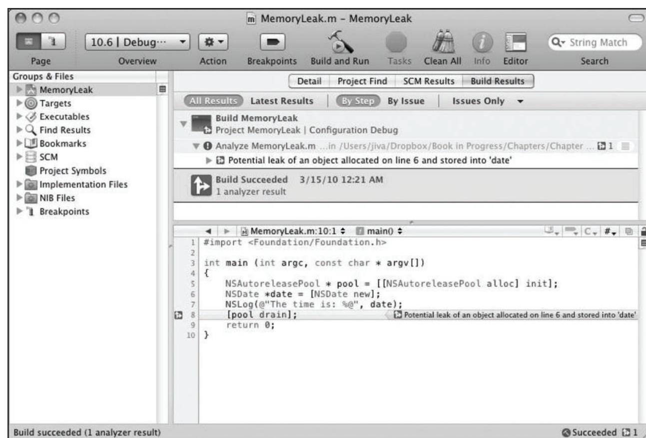


图 1-13　静态分析器输出

注意，该错误在屏幕上方的编译结果面板以及代码中显示。Clang 静态分析工具不仅可以捕捉到大多数编译器捕捉不到的错误，同时它的输出也比大多数编译器错误更清晰。如果你展开编译结果中错误消息对应的展开图标，就会显示和该错误相关的两条单独的信息。单击其中的一条消息，会显示内存分配的准确位置以及分析时的程序流程。你可以通过箭头来遍历代码。图 1-14 显示了错误展开后以及分析器完整输出显示后的相同代码。

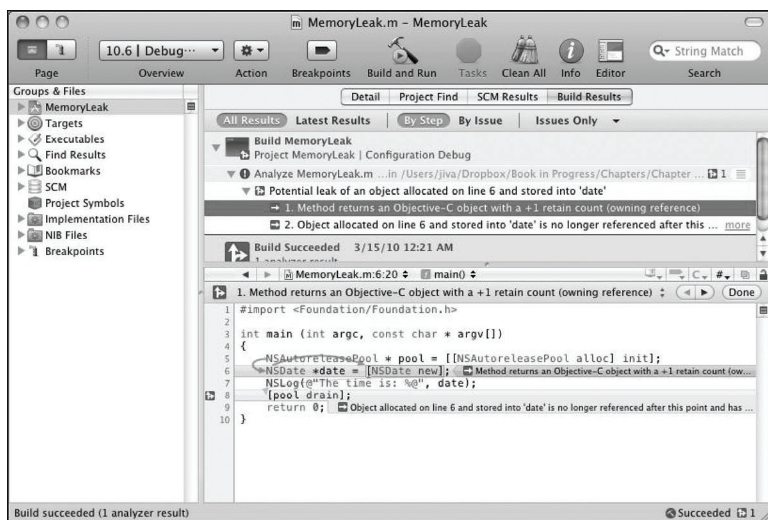


图 1-14 分析器的详细输出

按照代码清单 1-5 修改代码后，错误消失并且编译通过。

代码清单 1-5 修正后的代码

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSDate *date = [[NSDate alloc] init];
    NSLog(@"The time is: %@", date);
    [date release];
    [pool drain];
    return 0;
}
```



说明

重新编译应用后，Clang 静态分析器就不再标记错误了。

1.4　Objective-C 运行时

理解 Objective-C 最基本的一点是，Objective-C 是一种带有动态运行时的编译型语言。这就意味着该语言可以通过编译器编译，支持静态、编译时的类型检查，同时还可以链接到支持方法动态调度的运行时。利用动态运行时可以进行很多只有脚本语言可以做到的事情，比如“鸭子类型”和对象自省。



说明

鸭子类型指的是语言类型安全的一种类型，该类型假定如果一个对象“看起来像是鸭子，叫声像鸭子，就一定是鸭子”。这和静态类型是相对的。对于静态类型的情况，为了使得该语言可以解析其方法，对象必须是所声明的类型。两种技术都有各自的优缺点，但对于 Objective-C 来说，鸭子类型使得该语言具备了很多很酷的功能。

实际上主要有两种 Objective-C 运行时：64 位机上和 iPhone 上使用的“现代运行时”，以及在 32 位 Mac OS X 及其他地方使用的“遗留运行时”。“现代运行时”中有一些很有利于开发的特性，但由于未广泛使用，本书所使用的代码示例都是针对遗留运行时的。现代运行时完全向后兼容遗留运行时，所以不会对编写代码造成任何问题。在现代运行时环境有明显优势的地方，我会通过文字提示以供参考。

在编译应用的时候，Objective-C 运行时就会自动添加到应用中。除了需要使用一些高级功能的情况，使用它是完全透明的。到目前为止，需要理解的就是运行时在处理动态类型和静态类型方面的能力。这是 Objective-C 相对于其他编程语言的一个特色。

这种能力的一个副作用就是 `id` 数据类型的引入。`id` 数据类型是 Objective-C 中一种特殊的对象类型。`id` 类型的变量可以存放任意类型的对象。第 3 章中会详细介绍。

1.5　小结

本章介绍了 Xcode 集成开发环境，这在本书中开发应用时始终会用到。同时介绍了如何按实际需要配置 Xcode 集成开发环境，如何组织其中的文件，如何编译应用以及如何查看输出以找出代码中的错误。此外还介绍了编译过程、应用包的格式，同时简单解释了一下 Objective-C 运行时。

第 2 章

基本语法

本章概要

- ❑ 编写你的第一个程序
- ❑ 声明变量
- ❑ 使用函数
- ❑ 使用流控制语句
- ❑ 使用循环

在本章中，我将展示如何编写一个简单的 Objective-C 程序。这是一个会在控制台输出一条简短信息的非常简单的命令行程序。通过这个简单的程序来介绍一些 Objective-C 的基础知识。从实际写代码开始，再到使用变量和函数，最后，通过使用条件语句和循环来控制程序流。这些概念是学习编程语言的基础，在进入下一章之前你应该完整学习完本章的内容。

继续在第一章中所创建的 Xcode 项目中输入代码清单 2-1 所示的代码。这些代码应该写到按照项目名命名的源文件中，该文件位于源文件组中。

代码清单 2-1　你的第一个程序

```
#import <Foundation/Foundation.h>

int main(int argc, const char *argv[])
{
    NSLog(@"Hello from Objective-C");
    return 0;
}
```

讲解代码时我会稍微跳跃点，因为通过这种方式更容易解释。

我们就从第 3 行开始，这是 main 函数的声明。所有的 Objective-C 的应用都有一个 main 函数，通常你不用看它，因为它通常是在创建项目时由项目模板创建的。当编写图形应用程序时，几乎不需要编辑这段代码。之所以在这里展示它，是因为我还想教你如何编写命令行应用程序。

所有的 Objective-C 应用都有一个 main 函数。main 函数是程序开始也是程序结束的地方。程序已从操作系统调用 main 函数开始执行。argc 和 argv 这两个参数包含了通过命令行传递给应用程序的参数。程序接着开始逐行执行 main 函数中的每一行代码，直到遇到返回语句。在

本例中，返回语句直接返回 0。这表明程序成功地退出了。



说明

什么是函数？函数本质上是程序中的一个子程序。它是执行某项任务并返回值的代码分支。调用函数的时候，通过参数传送数据给函数（参数位于括号内），你可以通过几种方式从函数取回数据，但是主要的方式是通过接收返回值，即从函数返回到调用代码的值。

第 5 行调用了一个名为 `NSLog` 的函数。这个函数在程序运行时将传递给它的字符串输出到控制台。

第 4 行和第 7 行的大括号表示 `main` 函数的作用域，这是很常见的表示方法。第 5 行和第 6 行结尾的分号表示这些语句结束，编译器用分号将一个语句和另外一个区分开来。Objective-C 中的语句可以分成多行书写，因此，编译器在一个语句结束的时候需要一些标记，这样它才能够分析那条语句。什么时候在一行的结尾需要分号会困扰很多初学者。记住，除了流控制语句以外，函数内部的所有语句，都需要以分号结尾。另外，声明也需要以分号结尾。第一行是一个 `import` 语句，导入语句允许你在当前文件中加载另外一份文件的代码。在本例中，我们包含了 Foundation 框架的接口声明。（你可以通过本书的第二部分更多地了解 Foundation）。这行代码是必需的，有了它加载的代码，我们才能在第 5 行调用 `NSLog` 函数。看着有点怪的`@"Hello from Objective-C"` 就是我们所说的字符串，字符串就是代码中的文本，它们一般存储在程序的一个变量中并且可以在以后访问。在这里，这个字符串将作为参数传送给 `NSLog` 函数，然后在控制台显示。

编译并运行这个应用程序然后查看输出结果，你应该看到如图 2-1 所示的内容。

图 2-1　应用程序的输出

2.1 使用语句和表达式

所有的 Objective-C 程序都是由语句和表达式组成的。语句是用于执行某一操作的代码行。一般来说，语句没有返回值，因此不会改变当前行执行代码的状态，除非调用语句会导致执行其他操作。换言之，语句可以导致跳转到另外一行代码，并且那行代码会导致执行特定操作（如输出一些东西到控制台或者显示一个对话框），但它不会给该语句所在行的代码返回值。

而表达式则需要给调用代码返回一个值，因此可以用来改变程序流。

在大多数情况下，两者之间的区别小到可以忽略不计。在本书中，我交替使用这两个术语。

2.1.1 声明变量

你已经了解到编写一个简单的 Objective-C 程序所需的基础知识，现在你可以更进一步，按照代码清单 2-2 所示修改代码。

代码清单 2-2　一个带有变量的程序

```
#import <Foundation/Foundation.h>

int main(int argc, const char *argv[])
{
    int aVariable = 5555;
    NSLog(@"%ld", aVariable);
    return 0;
}
```

新增的代码展示出下一个概念——变量。

正如你所看到的，我添加了一个新的变量并且赋给它一个值，然后调用 `NSLog` 输出该变量。变量用来存储数据。变量有相应的为它分配的内存，可以用于存储想要存储的东西。在这里，我在 `aVariable` 变量中存储了 `5555`，然后将该变量传给 `NSLog` 函数。有趣的是，从一开始程序中就已经有了一些其他变量，也就是 `argc` 和 `argv`。和变量 `aVariable` 一样，它们存储的值可以在该程序的其他地方使用。例如，可以按代码清单 2-3 所示更改代码。

代码清单 2-3　使用 `argc` 变量

```
#import <Foundation/Foundation.h>

int main(int argc, const char *argv[])
{
    NSLog(@"The argument count is: %ld", argc);
    return 0;
}
```

在本例中，修改后的程序输出传给它的参数的个数。`argc` 变量存储传入程序的参数个数。通过向 `NSLog` 函数传递一个变量就可以将其输出。

变量可以在一个给定的作用域（也称栈帧）内声明，也可以在所有的栈帧之外声明（这种情

况下，它们就是全局变量）。栈帧在代码中通过大括号定义。例如，参见代码清单 2-4。



说明

通常认为使用全局变量是一个不好的编程习惯，应该避免。

代码清单 2-4　不同的栈

```
#import <Foundation/Foundation.h>

int main(int argc, const char *argv[])
{
    //这是第一个栈帧
    int aVariable1 = 5;
    if(aVariable1 > 4)
    {
        //这是第二个栈帧
        int aVariable2 = 10;
        NSLog(@"aVariable1: %ld", aVariable1); //正确
        NSLog(@"aVariable2: %ld", aVariable2); //正确
    }

    NSLog(@"aVariable1: %ld", aVariable1);  //正确
    NSLog(@"aVariable2: %ld", aVariable2);  //错误，aVariable2 这时已经不存在

    return 0;
}
```

在上面的代码中，变量 aVariable1 定义在第一个栈帧中。这个栈帧一直存在到第二个大括号的结束。我们会说“aVariable1 在第一个栈帧里面有效”，变量 aVariable2 只在第二个栈帧内有效（即第 7 行到第 12 行）。因此，当程序执行到第 15 行时，aVariable2 已经不复存在了，这就将导致错误。我们说“aVariable2 在 12 行第二个栈帧的结尾处越界了”。本章前面提到变量存储内存中的数据，它们所使用的内存可以在栈上分配，就像在这个例子中看到的那样。与这些变量相关联的内存在变量出了作用域后也一并释放了。于是，这时变量就不存在了。然而，变量的内存也可以在“堆”上分配，堆就是一个内存池，应用程序可以自己分配其中的数据，并且可以更大限度地控制它。然而，正如《蜘蛛侠》（*Spiderman*）中说的一样，“能力越大，责任越大”，你必须确保释放任何在堆上分配的内存。代码清单 2-5 展示了一个在堆上分配对象然后释放的例子。

代码清单 2-5　在堆上分配内存

```
//分配内存
SomeClass *aVariable = [[SomeClass alloc] init];

//操作变量 aVariable

[aVariable release]; //释放内存
```

这份代码清单展示了一个后面将花费大量时间介绍的概念：对象。现在，只要知道变量 `aVariable` 在堆上分配内存即可。这也意味着如果我们在代码的结尾处不释放这部分内存，它无法被其他变量使用，这也就是我们常说的*内存泄漏*。判断一个变量的内存是分配在堆上还是栈上的一个关键方法是看它是不是一个指针，在前面的例子里，`aVariable` 声明中的`*`操作符表明这是一个指针。指针是一个指向内存地址的变量。

请注意，到目前为止我们用过的所有变量的名字前都没有`*`操作符。这意味着，它们的内存都是在栈上分配的。虽然你也可以使用仅在栈上分配的指针，但通常你看到的 Objective-C 的指针都是在堆上分配的变量的指针。关键是寻找一种类似这里所示的内存分配函数。

指针是一种很难理解的概念，为此后面将用一节详细介绍指针。

本章前面介绍了全局变量。同样，我们用局部变量来描述在栈上分配的变量。*局部变量*的存在范围仅限于局部作用域。*局部作用域*是“当前栈帧”的另一种叫法。还有一些其他类型的变量，包括*成员变量*（这种变量是一个类的成员）和*实例变量*（这种变量用来存储特定的对象实例）。我会在第 3 章介绍对象的时候介绍这两种变量。

在 Objective-C 中使用变量时，必须先声明，这意味着你必须先告诉编译器你将使用它们。在声明变量时就必须给定类型。类型可以分为 3 大类：标量、指针和结构体。我们在下一节介绍这 3 种类型。

2.1.2 使用注释

代码清单 2-4 和代码清单 2-5 中介绍了另外一种语法，双斜线。在 Objective-C 中，`//`后面的语句是注释。它们会被编译器忽略掉，可以在注释中包含任何内容。通常，它们用来在代码中给出一些可读的文档。但是经常也可以用来暂时删除代码或者使代码更好看。重要的是要知道不管在代码中的何处看到`//`，那么该行的所有内容都会被编译器忽略。

除了使用`//`风格的注释外，还有一种通过`/*`和`*/`表示的注释语法。这种情况下，注释就不再是到行末结束，它仅注释`/*`和`*/`之间的文本。

使用哪一种注释方式完全取决于你。我比较喜欢`//`风格的注释，所以本书通篇都将选用它来注释。

2.1.3 标量类型

最基本形式的变量是标量，*标量*是一次只能存储一个值的变量。整数、浮点数和字符都是标量。标量有不同的预定义内存空间和可以存储的值的大小。在决定用什么类型定义变量之前应该知道这些类型的限制。表 2-1 展示了 Objective-C 中经常用到的标量类型的值域。

这里大多数的标量可以在 C 和 C++中通用，但是也有一些是 Objective-C 所独有的。

苹果公司的操作系统和库对 32 位和 64 位架构一直都支持得很好。然而从编程的角度看，这可不是一件容易事。例如，有些值（如数组索引）会受益于增加的 64 位整数的上限。因此，在理想的情况下，我们希望代码可以在 32 位平台和 64 位平台之间完美地转换，因此，苹果公司提

供了 NSInteger 和 NSUInteger 类型，它们可以根据我们的编译平台的架构自动在 32 位或 64 位之间转换。

表 2-1　常用标量类型

类　型	描　述
int	+/– 2 147 483 647 之间的整数值
unsigned int	0 和 4 294 967 296 之间的整数值
float	+/– 16 777 216 之间的浮点值
double	+/– 2 147 483 647 之间的浮点值
long	根据芯片架构的不同，大小从 32 位到 64 位的整数值
long long	64 位整数
char	单个字符，通过整型表示
BOOL	布尔值，YES 或者 NO
NSInteger	在 32 位机器上，和 int 相同。在 64 位机器上，范围是+/– 4 294 967 296
NSUInteger	在 32 位机器上和 unsigned int 相同

标量使用起来非常简单。要声明一个标量，你只要告诉编译器变量的类型和名字即可。也可以给它一个初始值。例如，代码清单 2-6 显示了如何声明一些常见的标量变量。

代码清单 2-6　声明标量变量

```
int foo = 10;
double bar = 500.0;
float baz;
unsigned long n;
NSInteger x;
char a = 'a';
```



说明

注意变量 a 被初始化为'a'。使用单引号包含一个字符是告诉编译器将其值当做 char 型。不要把它和我们后面将介绍的字符串混淆，字符串是通过@""来指定的。

2.1.4　使用特殊变量修饰符

除了变量类型和变量名之外，还有一些关键字用来修饰你声明的变量类型。其中最重要的修饰关键字是你将在本书中看到的 static 和 const。

正如我前面提到的，声明局部变量时，变量的内存通常在每次程序进入该局部变量的作用域时分配并在离开时释放。这类存储称为自动存储，或者通过默认修饰符关键字 auto 修饰。

static 关键字会在声明变量的时候分配内存，所以在程序运行期间只会分配一次内存。之后在程序中访问该变量时，实际上都是在访问原先分配的内存。这一点很重要，因为这样你就可以指

定一个局部变量来长期保存其中的值。这适用于存储创建时使用大量的资源，并且不常改变的局部变量。代码清单 2-7 展示了如何在函数中使用 `static` 关键字来优化初始化变量之类代价高的操作。

代码清单 2-7 使用 static

```
void someFunction()
{
    //不论你调用多少次
    //x 只会创建并初始化一次
    static Expensive *x = [[Expensive alloc] initWithData:...];
    //操作 x
    [x doSomeOperation];
}

int main(int argc, char *argv[])
{
    someFunction(); //x 在 someFunction 中创建
    someFunction(); //x 已经存在，不会再次创建
    return 0;
}
```

因为全局变量默认位于全局作用域中，所以其行为和静态变量一样。也就是说，它们只分配一次内存，并在整个程序运行期间保持其值不变。将 `static` 关键字应用到一个全局变量时，它会修改全局变量的作用域，从而只能在声明该变量的文件内部访问该变量。这和通常的全局变量有很大不同，全局变量的作用域是程序的任何地方。

`static` 关键字被认为是存储修饰符。在 Objective-C 中有好几个这类修饰符。`register` 修饰符用来提示编译器所存储的数据将会经常被访问，因此适合存储在 CPU 的寄存器中。这个关键字很少用到。另外一个更常用的关键字是 `extern`。该修饰符表示所定义的函数或声明的变量引用了应用程序另一个编译单元中定义或者分配的实际变量或者函数。本章后面讲到函数的时候你就会看到 `extern` 关键字的使用。

`const` 关键字同样会修改所声明变量的内存行为，但是这种情况下，变量是只读的。这意味着变量被初始化以后，它的值将不能改变。这在声明不能修改的变量（比如常量）时非常有用。将此类变量声明成 `const`，编译器将强制这种行为。如果后面你不小心尝试修改这样的变量，这将是个 bug，编译器将会产生一个错误。代码清单 2-8 展示了如何使用 `const` 关键字避免常量被改写。

代码清单 2-8 使用 const 关键字

```
int main(int argc, char *argv)
{
    const NSString *foo = @"MY_CONSTANT";

    //执行一些操作

    foo = @"SOME_OTHER_VALUE"; //这会产生一个编译器错误

    if([foo isEqualToString:@"MY_CONSTANT"])
    {
```

```
            //执行操作
        }
    }
```

2.1.5　结构体

结构体由 `struct` 表示，是一种可以包含多个子变量的自定义类型。例如，如果你想声明一个变量，将 *x* 和 *y* 坐标组合在一起来表示一个点，可能会使用 `struct` 来声明这个变量。这可以利用 `struct` 关键字实现。

声明一个 `struct` 需要两步。首先，你必须先告诉编译器结构体本身，然后使用这个结构体来声明一个类型为已经定义好的结构体的变量。继续这个声明一个变量来表示点的例子，代码清单 2-9 展示了最初如何定义结构体。再次说明，这是该过程的第一步——定义结构体。

代码清单 2-9　定义一个结构体

```
struct Point
{
    float x;
    float y;
};
```

本例中我定义了一个 `Point` 结构体，它包含 `x` 和 `y` 这两个浮点型的成员变量。现在，在定义点的结构体后就可以声明变量了，它最终会保存你要使用的实际的点。实现过程如代码清单 2-10 所示。

代码清单 2-10　声明结构体实例

```
struct Point p;
```

结构体的成员也可以是其他结构体。例如，代码清单 2-11 展示了通过两个点定义一条直线的结构体。

代码清单 2-11　复合结构体

```
struct Line
{
    struct Point start;
    struct Point end;
};
```

将这些放在一起，就可以展示实际使用一个结构体来存储和显示一些点的代码，如代码清单 2-12 所示。

代码清单 2-12　使用点结构体

```
#import <Foundation/Foundation.h>

//声明点结构体
```

```
struct Point
{
    float x;
    float y;
};

int main(int argc, const char *argv[])
{
    //声明点变量
    struct Point p;
    //给结构体成员赋值
    p.x = 20.0;
    p.y = 50.0;

    //之后使用该点
    moveCursorToPoint(p);

    return 0;
}
```

Objective-C 和 Cocoa 使用结构体来存储点、矩形等。使用结构体的好处在于这是一种轻量级的存储相关变量组的方法。第 3 章将介绍如何使用对象来组合相关的变量以及操作这些变量的方法。不过，对象有相对应的开销。而结构体除了创建构成它的成员变量外没有其他开销。因此在对性能比较敏感的地方，不适合使用对象。

2.1.6　使用类型定义

每一次想定义一个点的时候都要输入 `struct Point`，自然你很快就会觉得单调乏味。幸好，Objective-C 为此提供了另外一种结构体可以帮助我们，这个结构体就是 `typedef`。

`typedef` 这个词来自 type definition（类型定义），从本质上支持定义你自己的类型。和结构体一起使用可以更好地定义一个自定义类型来表示你的结构体。你可以在通常使用结构体定义的任何地方使用这个自定义类型。代码清单 2-13 展示了使用 `typedef` 的 `Point` 结构体的例子。

代码清单 2-13　使用点结构体和类型定义

```
#import <Foundation/Foundation.h>

//声明点结构体
typedef struct
{
    float x;
    float y;
} Point;

int main(int argc, const char *argv[])
{
    //声明点变量
    Point p;
    //给结构体成员赋值
```

```
    p.x = 20.0;
    p.y = 50.0;

    //使用这个点
    moveCursorToPoint(p);

    return 0;
}
```

正如你所看到的，简单地通过在 struct 关键字前加上 typedef 关键字，并将结构体的名字移到结构体定义的结尾，就实现了很神奇的事。这样你就可以简单地通过 Point p 声明变量 p。事实上 Point 成为了“一等”类型，你可以在使用任何其他类型的地方使用它。

使用 typedef 关键字的语法是：typedef *变量定义 新类型名*。其中*变量定义*是你使用新类型时想要插入的实际类型。*新类型名*是你将在程序中使用的新类型的名字。

由于类型定义支持使用类型名来更清晰地描述将要存储在变量中的数据的类型，所以类型定义是一个很好的补充，让你的代码比没有使用类型定义的代码更可读。第 5 章介绍代码块的时候还会涉及类型定义的使用。

2.1.7　使用enum

定义自定义数据类型的另一种方式就是使用 enum 关键字，enum 是枚举类型的简称，利用枚举类型你可以创建一种数据类型，用于存储一个有限的可能值列表。在 Cocoa 和 Cocoa Touch 中，当可能值列表属于某个有界集时经常使用枚举作为参数和返回值。

要定义一个枚举类型，需要使用 enum 关键字，随后是要声明的枚举的标签，然后是{}，其中包含了用逗号隔开的可能值的列表。代码清单 2-14 展示一个枚举定义。

代码清单 2-14　创建枚举类型

```
enum MyEnum
{
    Value1,
    Value2,
    Value3
};
```

要在代码中使用枚举类型，必须声明一个枚举类型的变量，之后跟有定义枚举时指定的标签。然后赋一个值给它，你可以直接使用枚举定义中的值。该实现过程如代码清单 2-15 所示。

代码清单 2-15　使用枚举

```
enum MyEnum foo;
foo = Value1;

//或者函数
enum MyEnum myFunction();

//作为函数参数
void myFunction(enum MyEnum foo);
```

枚举本身的实际值由编译器来决定，但是它们默认是整型。第一个值为 0，第二个值为 1，以此类推。你可以强制给一个枚举成员赋一个特殊的数值，只要在枚举定义的时候提供那个值即可，如代码清单 2-16 所示。

代码清单 2-16　给枚举成员赋值

```
enum MyEnum
{
    Value1 = 20,
    Value2 = 13,
    Value3 = 155
};
```

当原有代码中需要一个特殊数值时这非常有用。

每次使用枚举时都输入 enum 是很麻烦的事情。所以，和结构体一样枚举也可以进行类型定义。实现过程如代码清单 2-17 所示。

代码清单 2-17　枚举类型的类型定义

```
enum MyEnumType
{
    Value1,
    Value2,
    Value3
};
typedef enum MyEnumType MyEnum;

//现在你可以输入……
MyEnum foo;
foo = Value1;
```

Cocoa 和 Cocoa Touch 大量使用枚举。枚举的好处是它支持编译时检查传给函数的参数是否是一个有限集中的某个值。如果你不小心传入了一个错误的值，编译器就生成一个错误。

2.1.8　指针

我将要介绍的第三种类型的变量是指针。指针是一个能让你晕头转向、难以理解的概念，不过幸好，在 Objective-C 中很少用到指针的复杂功能。不过理解这些还是很重要的。

回顾一下，典型的变量将数据存储在内存中。计算机通过地址在内存里寻找变量。打个比方，如果将变量想象成一个人（数据）居住的房子，那房子的街道地址就是变量的地址。

指针是一个包含了另一个变量的地址的变量。声明一个指针和声明一个你想指向的任意类型的变量类似，除了声明中要包含指针操作符。指针操作符是一个星号（*）。你可以用取址操作符（&）获取一个变量的地址。

代码清单 2-18 展示了一个声明整型指针的例子。

代码清单 2-18　声明指针

```
#import <Foundation/Foundation.h>

int main(int argc, char *argv[])
{
    int x = 5;
    int *y = &x;

    NSLog(@"X:%ld - Y:%ld", x, y);
    return 0;
}
```

在这段代码中，我们声明了一个整型变量 x。在该变量中存储了 5。然后我们声明了另外一个变量 y，这是一个指向整型的指针，我们用它来存储 x 的地址。

指针可以当做普通变量使用，但是当你直接访问它时，就像这个程序的第 8 行那样，你获取到的是内存地址。在本例中，运行程序，会看到如图 2-2 所示的输出结果。

你的 y 值将可能会和我的不同，因为你的计算机可能将 x 值存储在一个和我不同的地址。这很正常。星号操作符（ * ）也是指针取值操作符。指针取值后你可以访问指针所指向的变量的值。换言之，你想要打印出指针所指向的实际值，也就是 x 值，那么应该按照代码清单 2-19 修改程序。

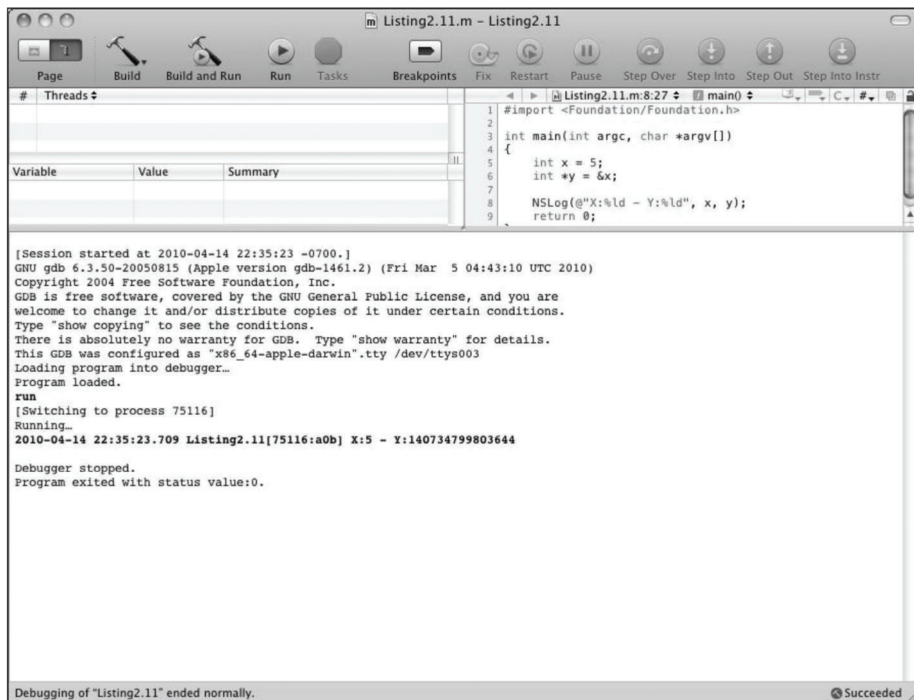


图 2-2　指针程序的输出

代码清单 2-19　对指针取值

```
#import <Foundation/Foundation.h>

int main(int argc, char *argv[])
{
    int x = 5;
    int *y = &x;

    NSLog(@"X:%ld - Y:%ld", x, *y);
    return 0;
}
```

注意，本次我们对 y 取值，这意味着运行这个程序的时候，你将会看到如图 2-3 所示的结果。

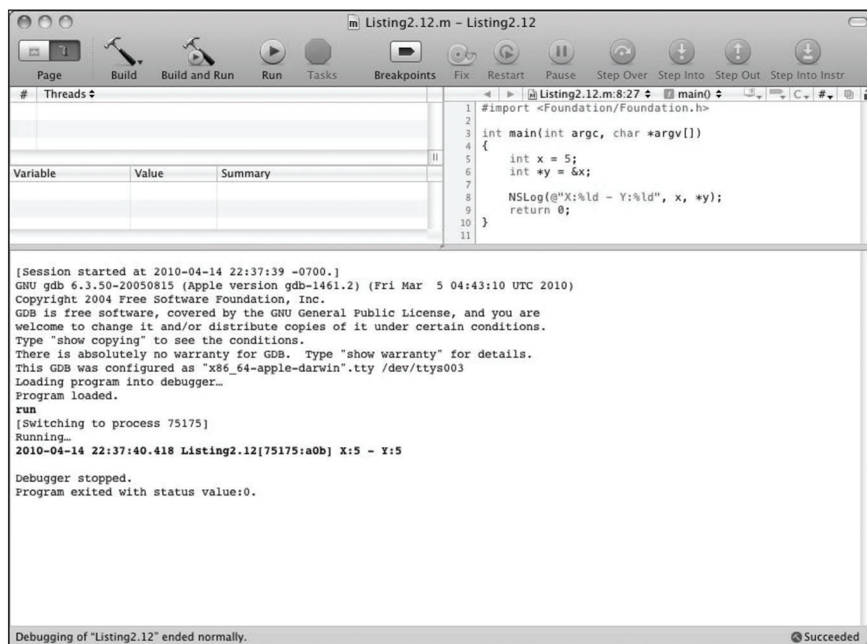


图 2-3　取值示例代码的输出

有趣的是指针可以和常规的变量一样操作。这意味着你可以对它们进行递增，也可以递减，可以加上一个值它，也可以减去一个值。这些事情你都可以做，但是你所做的实际是改变指针在内存里所指向的地址。因此，你可以对新地址的指针取值，获取一个不同于最初指向值的新值。

这里介绍指针的大部分内容都和高级主题相关。在日常的 Objective-C 中，基本不需要对很多指针取值。（这个规则有两个例外，在学习本书的过程中我会指出。）要深入理解底层指针，我建议找一本好的 C 语言编程书。

Objective-C 指针典型的使用主要是声明对象。Objective-C 中的对象实际上是指针。幸好在

很大程度上，即使它们是指针，你也没必要把它们看做指针。重要的是要记得，任何时候你声明的任何对象都要像指针一样声明，如代码清单 2-20 所示。

代码清单 2-20　对象的指针

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    NSString *foo = [NSString stringWithString:@"Foobar"];

    NSLog(@"foo: %@", foo);

    [pool drain];
    return 0;
}
```

如果编译并运行这个程序，将看到和图 2-4 所示内容类似的结果。

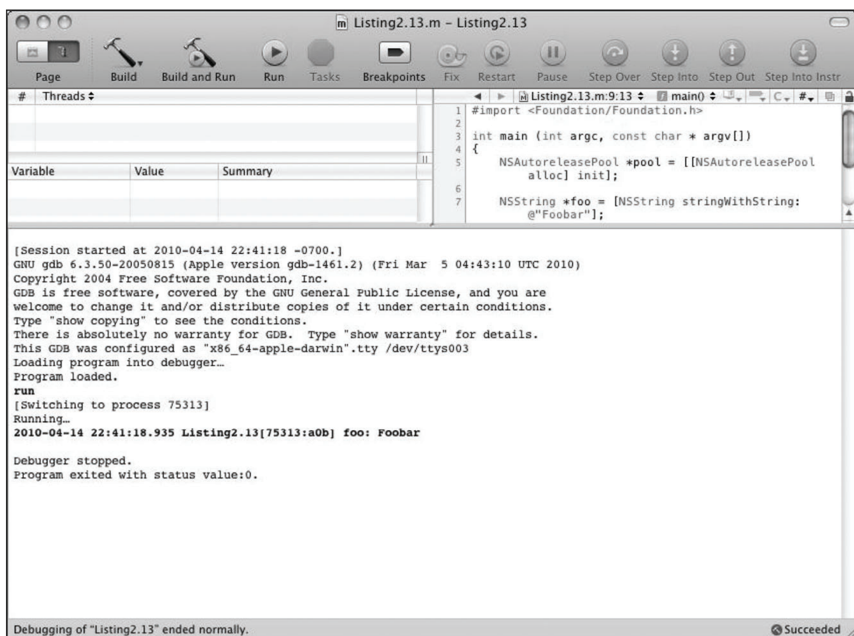


图 2-4　对象指针输出示例



说明

忽略和 NSAutoreleasePool 有关的代码，后面的章节会介绍。

另外还有一种不常见的指针的用法，你将在本书介绍对象指针的使用时遇到，例如使用 NSError 对象从失败的函数调用获取一个错误信息。在第 10 章讨论这部分时，我会更详细解释。

2.1.9　使用运算符

和数学一样，编程语言通常支持运算符。运算符就是“操作”变量和值的函数。例如，表达式“5+4”包含了值 5 和 4 以及运算符+。这种情况下，+操作符可以称作双目运算符。所以，它对两个值进行操作，5 和 4 一左一右。另外一种类型的运算符是单目运算符。单目运算符只能对一个值进行操作。例如，取址运算符&就是单目运算符。它用于获取操作值的地址。

表 2-2 展示了 Objective-C 中大多数常见的运算符。

表 2-2　常见的运算符

运　算　符	作　　用
() [] -> .	圆括号、数组操作符、取值
! ~ − + * & ++ −−	非、加/减（一元）、取值（一元）、取址（一元）、递增（一元）、递减（一元）
* / %	乘法、除法、取模（二元）
<< >>	移位（二元）
< <= > >=	比较运算符（二元）
== !=	比较运算符（二元）
&	按位与（二元）
^	按位异或（二元）
|	按位或（二元）
&&	逻辑与（二元）
||	逻辑或（二元）
= += −= *= /= %= &= |= ^= <<= >>=	赋值操作（二元）

通常，运算符返回某种结果，结果将赋给某个变量或用在控制语句中来改变程序流。例如键入代码清单 2-21 所示的示例程序并查看结果。

代码清单 2-21　使用运算符

```
#import <Foundation/Foundation.h>

int main(int argc, char *argv[])
{
    int a = 10;
    int b = 3;
    int c = a + b;
    int d = a - b;
    int e = a * b;
    int f = a / b;
    int g = a % b;
```

```
    NSLog(@"a: %ld", a);
    NSLog(@"b: %ld", b);
    NSLog(@"c: %ld", c);
    NSLog(@"d: %ld", d);
    NSLog(@"e: %ld", e);
    NSLog(@"f: %ld", f);
    NSLog(@"g: %ld", g);

    return 0;
}
```

在这段代码中，我们将值 3 和 7 分别赋值给 a 和 b，然后我们用+运算符把它们加到一起，用–运算符做减法，以及乘除操作等。运行程序并查看如图 2-5 所示的输出。

注意，程序输出了每一个数学运算的结果。我在程序中动了一点小手脚。如果你留意，f 的值是错误的。这是因为 10 除以 3 的结果不是一个整数，是一个分数或者从计算机的角度来说是一个浮点数。通常你应该将它存储为浮点型或双精度型，两者都能够表示浮点值。在这里我将它存储到整型中，这意味着小数位的值会被截断，导致结果只有整数部分的值。即使你用浮点数存储结果，仍可能从除法操作中得到截断的值，因为操作数都是整型。这里附带说一下你要记得使用整型数时，应该在执行任何可能出现浮点值的数学运算前，先将操作数转换成浮点值。下一个值 g 示范了取模操作符，它返回除法运算的余数。所以，在需要使用分数，而且是用一个整数和余数表示的情况下，就可以使用这种技术。f 和 g 可以合在一起来表示 3 和余数 1。

图 2-5　运算符程序的输出

运算符有优先级，就和在数学里一样。这个优先级如表 2-2 所示。有时候很难准确记住优先级的顺序，为避免混乱，你应该设法记住圆括号的优先级最高。它们是可以随意在表达式里添加使表达式更清晰的句法糖（syntactical sugar）。请根据情况使用。

2.1.10 三目运算符

除了我已经介绍的单目和双目运算符，还有一种三目运算符。这个算符是`?:`，它可以用来替代 `if/else` 控制语句。我不在这里介绍这个运算符，因为它放在 2.3 节介绍更合适。因此，更多信息请参见 2.3 节。

2.2 使用函数

到目前为止，我已经介绍了该如何将指令发送给计算机，从而让计算机在执行程序时顺序执行这些指令。然而，随着程序大小和复杂度的增加，你很快就会发现把所有的指令一个接着一个按顺序放入程序中会变得非常费力。此时你希望能复用一部分代码，不过复制粘贴是一种非常糟糕的代码复用方法。

幸好你可以使用几种机制来解决这两个问题。我将在本节介绍第一种机制，也就是使用函数。在面向对象编程之前，将程序分解成更容易复用的小块的首选方法就是面向过程编程。

Objective-C 是一种完全面向对象的编程语言，在应用开发过程中大多数情况下会使用面向对象编程。但是，面向过程编程是一种你需要理解的重要的基本构建块，因为 Objective-C 仍有一部分本质上是面向过程的。

2.2.1 函数

你会问的第一个问题可能是“什么是函数？”。函数从本质上来说是一个在应用程序中声明子程序的方法。利用函数你能够将一部分程序指令封装成某种可以被命名并且在需要时可以在程序任何地方使用任意次的东西。

看个例子，代码清单 2-22 中的程序用于计算 5 的阶乘（它应该产生一个值 120）。它使用了一个 `for` 循环来进行计算，这是下一节要介绍的一个概念，不过现在你可以忽略它，只要试着将其看做一种计算即可。

代码清单 2-22 阶乘计算

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
    int a = 5;
    int result = 1;

    for(int i = 1; i <= a; ++i)
```

```
        {
            result = result * i;
        }

        NSLog(@"%ld", result);

        return 0;
    }
```

现在，这个程序有趣的地方在于，为了进行这种计算我们做了很多工作。如果需要改变计算阶乘的因数该怎么做？可以抽出一部分代码到函数中，这样就可以多次复用了。或许可以创建一个接收一个整型参数的函数，然后返回这个整型数的阶乘。

代码清单 2-23 展示了这样的一个程序。

代码清单 2-23　提取计算到一个函数

```
#import <Foundation/Foundation.h>

long int calculateFactorial(int value)
{
    long int result = 1;

    for(int i = 1; i <= value; ++i)
    {
        result = result * i;
    }

    return result;
}

int main (int argc, const char * argv[])
{
    int a = 5;
    long int result = calculateFactorial(a);

    NSLog(@"%ld", result);

    return 0;
}
```

主函数中之前包含的计算阶乘的代码，现在换成调用 `calculateFactorial` 函数。存储用于计算的值的变量是在主函数的局部作用域内声明的，因此函数 `calculateFactorial` 看不到它。为了使 `calculateFactorial` 函数可以从主函数获取这个值，需要通过函数调用将值传递给 `calculateFactorial` 函数。

编译并运行这个程序，输出和前面几乎一致。现在你可以复用这些代码来计算一组不同的值的阶乘，如代码清单 2-24 所示。

代码清单 2-24　使用函数的示例

```
#import <Foundation/Foundation.h>

long int calculateFactorial(int value)
{
    long int result = 1;

    for(int i = 1; i <= value; ++i)
    {
        result = result * i;
    }

    return result;
}

int main (int argc, const char * argv[])
{
    NSLog(@"5!: %ld", calculateFactorial(5));
    NSLog(@"10!: %ld", calculateFactorial(10));
    NSLog(@"15!: %ld", calculateFactorial(15));
    NSLog(@"20!: %ld", calculateFactorial(20));

    return 0;
}
```

从中可以看出，手动计算那些数值较大的数的阶乘会很耗时，计算机却可以很快完成计算。



说明

我使用 long int 来保存结果值是因为数值很快就会变大。

2.2.2　定义函数

要创建一个函数，必须定义它。定义函数的过程首先是告诉编译器函数的参数类型和返回值类型，然后将函数代码置于大括号中。

代码清单 2-25 再次给出了函数示例。注意，这里突出了函数定义中与返回值和函数参数相关的部分。

代码清单 2-25　函数详解

```
long int calculateFactorial(int value) //函数声明
{
    long int result = 1;

    for(int i = 1; i <= value; ++i)
    {
        result = result * i;
```

```
    }

    return result; //在这里返回值
}
```

函数定义的第一行称作函数签名。在程序中它必须是唯一的，这样编译器才能找到该函数。

定义一个函数时需要将传递给函数的参数放在函数名后的圆括号中。每一个参数通过类型和变量名指定。在有多个参数的情况下，不同的参数通过逗号隔开。这些参数和在函数内声明的变量类似，在该函数内可用。

返回值的类型在函数名的左边指定。作为声明的一部分，返回值没有一个与其关联的变量名。通过函数体内的 `return` 关键字定义函数的返回值。这应该是函数的最后一条语句。

在 Objective-C 中，参数和返回值通过 “传值”的方式传递。这意味着当你传递一个参数给函数时，运行时实际上会创建一个传入值的副本。这就意味着在函数内部改变这些值将不会影响到调用函数中的值。但是，如果你传递了一个指针，对指针变量的任何改动都会影响到调用函数中的原变量。

代码清单 2-26 展示了这个概念。

代码清单 2-26　使用指针

```
#import <Foundation/Foundation.h>

void myFunction(int a, int *b)
{
    a = 20;
    *b = 20; //反引用指针以访问原始值
}

int main(int argc, const char *argv[])
{
    int a = 10;
    int b = 10;

    myFunction(a, &b); //使用取址运算符把 b 转换为指针

    NSLog(@"a: %ld", a);
    NSLog(@"b: %ld", b);

    return 0;
}
```

同样，必须注意不要返回指向在函数退出时超出了作用域的变量的指针或者引用。例如，如果你返回函数内部的一个值的地址，当调用函数尝试访问这个值时，这个变量已经被释放，因此已不存在——这会导致程序崩溃。

第 3 章介绍对象和第 4 章介绍内存管理时，我将介绍如何返回函数内创建的指针到其他的函数。不过现在仅仅是简单的值。

2.2.3 实现与接口

函数的集合可以归组到一个单独的源文件中，这样主源码和所有编程逻辑就不会混在一起。这些源码文件通常称为单元。

当你将源代码分离到单元，一个单元由两个文件组成：其中一个包含了函数声明，另外一个包含了这些声明的定义。声明和定义之间的不同是一个必须理解的重要概念。

你已经看过函数定义了。到目前为止使用的函数都是完整定义的。*函数定义*是实际编写构成函数的代码，*函数声明*是声明函数的签名，这包括返回值类型、函数名以及参数。这不包括函数功能的实际定义。

函数声明可以说是声明函数的接口。这是一个不严格的定义，但后面介绍面向对象编程时，你将看到接口的概念被大量使用。

扩展一下这种思想，函数定义就可以说是所声明接口的实现。

要想声明一个可以在其他单元使用的函数，就要使用 `extern` 关键字并在后面跟上和函数定义类似的函数签名。因为函数声明是一条语句，就像声明变量一样，需要在函数签名的末尾加上一个分号。什么时候该加分号什么时候不加经常令程序员新手困惑不已。一定要记住，函数声明后要加分号而函数定义后面不加。

所以，想象一下我们想把阶乘计算方法封装到一个独立的单元中。

为此需要三步。第一步将为单元创建一个接口文件。通常，Objective-C 中的接口文件的扩展名是.h，这表示该文件是头文件。在 Xcode 项目创建一个新的头文件 Factorial.h，并加入如代码清单 2-27 所示的代码。

代码清单 2-27 函数声明

```
extern long int calculateFactorial(int value);
```

这是 `calculateFactorial` 函数的函数声明。下一步是创建一个实现文件。实现文件的扩展名是.c。在 Xcode 项目中创建一个新的 C 源文件 Factorial.c，并按代码清单 2-28 进行修改。

代码清单 2-28 函数定义

```
long int calculateFactorial(int value)
{
    long int result = 1;

    for(int i = 1; i <= value; ++i)
    {
        result = result * i;
    }

    return result;
}
```

最后，需要修改调用函数的文件以包含单元的接口文件。按代码清单 2-29 修改源代码文件。

代码清单 2-29　调用函数

```
#import <Foundation/Foundation.h>
#import "Factorial.h"


int main (int argc, const char * argv[])
{
    NSLog(@"5!: %ld", calculateFactorial(5));
    NSLog(@"10!: %ld", calculateFactorial(10));
    NSLog(@"15!: %ld", calculateFactorial(15));
    NSLog(@"20!: %ld", calculateFactorial(20));

    return 0;
}
```

函数的定义现在已经从这个文件移除并替换成一个导入命令。

导入语句用来在当前单元中包含其他单元的接口。导入语句的工作原理就是在项目目录以及项目相关的框架中查找需要包含的接口文件。使用尖括号<>来包含文件名时是通过系统路径搜索接口文件，使用 Foundation 框架就是一个实例。使用引号时就只会搜索项目的当前目录及其子目录。

接口的文件的搜索路径的规则可能很难记住，所以我建议你只需记住这两条规则：如果接口文件是你的项目的一部分，或是你创建的，在导入语句中指明时应该使用引号；否则，如果接口文件是一个第三方框架或苹果提供的框架的一部分，那么就应该使用尖括号。

2.2.4　链接实现文件

在将代码分成不同的单元并在代码中使用这些单元时，你还需要做一件事。必须确保实现文件被链接到可执行文件。可执行文件由通过 Objective-C 运行时链接在一起的多个实现文件组成。当添加多个实现文件到你的项目，就必须确定它们也被链接到可执行文件。通常，当你创建一个新文件时，Xcode 询问你是否要在项目中包含它。注意，图 2-6 中的复选框表示该文件将被添加到代码清单 2-6 的目标文件。

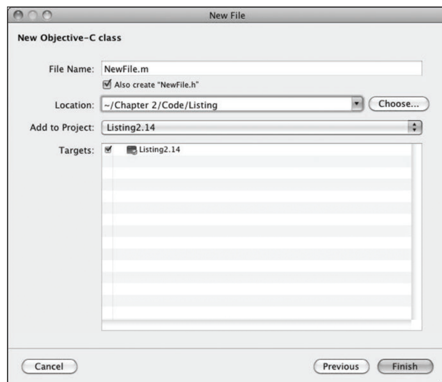


图 2-6　添加一个文件到目标文件

如果你忘记这样做，或者添加另外一个目标文件，并需要在目标文件中包含一个已经存在的实现文件，那么知道如何手动将一个已经存在的实现文件与当前的目标文件链接起来是很重要的。

为此只需选择想在当前目标文件中包含的可实现文件，单击编辑器顶部的 Detail 选项卡，并确保选中靶心列的复选框，如图 2-7 所示。

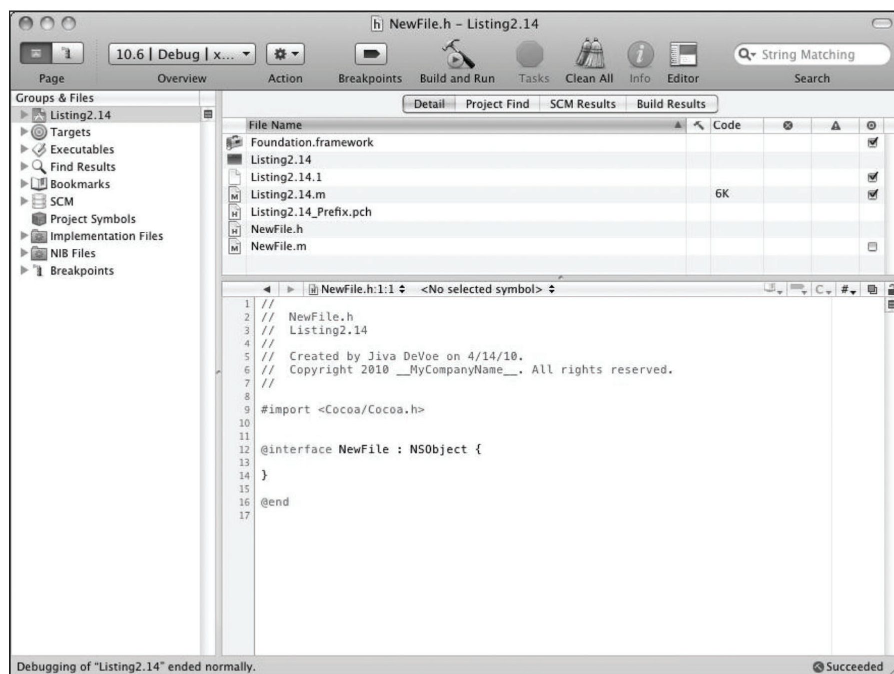


图 2-7　目标文件的复选框

再次强调，注意靶心复选框，NewFile.m 文件没有选中，要在目标文件中包含这个文件，选中这个复选框。

现在你知道一些语法和程序组织的基本知识，接着可以看看在 Objective-C 程序中如何控制程序流。

2.3　控制程序流

只能做一件事的应用程序是非常呆板的。如果不能根据条件在运行时做决定和分支，那么应用程序就没什么用。幸运的是，我们不用担心这些，因为 Objective-C 提供了一个包含丰富的流控制机制工具的工具箱。

这些流控制机制可以分为两大类。

第一类机制是条件控制语句，利用这些语句可以在运行时根据条件改变应用的执行路径。通

常可以在程序的中间做分支。程序走哪一条分支由变量决定，这些变量可以通过参数、用户输入或者其他你想要的条件来设置。

流控制机制的第二类是循环。循环支持重复执行一些操作集直到满足某个条件。利用循环可以遍历列表中的项，执行固定次数的操作，等等。

我们将在本章的剩余部分介绍这两类机制并看看如何在应用中使用这些机制来控制程序执行。

2.3.1　使用条件语句

重申一下，利用条件语句可以在代码中包含两个或更多不同的执行分支，条件语句有三种结构。

1. 使用 if-else

第一种结构是 if-else 结构。如代码清单 2-30 所示。

代码清单 2-30　if 语句

```
if (n > 75)
{
      NSLog(@"%ld is greater than 75!", n);
}
```

这个结构的基本语法包括一个 if 语句，后面跟着用小括号隔开的 if 语句的条件，然后是满足 if 条件语句时所执行的代码块。如果条件语句不满足，也可以提供一个可选的 else 代码块。这是在 if 语句后面提供的代码块，如代码清单 2-31 所示。

代码清单 2-31　带 else 的 if 语句

```
if (n > 75)
{
     NSLog(@"%ld is greater than 75!", n);
}
else
{
     NSLog(@"%ld is less than 75.", n);
}
```

else 语句也可以带一个和 else 语句在同一行的 if 语句。这时，如果最初的 if 语句为假将会执行 else 块，但是如果这时 if 语句为真，else 块将会被跳过。在这种情况下，你可以一个接着一个使用多个 else-if 语句。通常，应该在最后跟着一个 else 语句，当所有 if 条件都没有满足时执行，如代码清单 2-32 所示。

代码清单 2-32　if、else-if 和 else

```
 if (n > 75)
{
    NSLog(@"%ld is greater than 75!", n);
}
```

```
else if(n < 25)
{
    NSLog(@"%ld is less than 25!", n);
}
else
{
    NSLog(@"%ld is between 24 and 76.", n);
}
```

如果 if-else 语句后的代码块是一条语句，可以省略大括号。

if 或 else if 语句后的条件语句可以是任何返回布尔值 YES 或 NO 的语句。另外，对于返回值不是严格的布尔值的一些函数或语句，0 返回值被认为是假，或称为 NO，其他任何返回值则被认为是真，或称为 YES。



警告

可以放在 if 的条件语句中的任何语句，都可能导致一个在代码中非常常见的错误，在程序员想要使用==运算符检查相等性，但是他们不小心意外地使用了=运算符。这时，赋值操作将一直返回真。在你的代码中要非常注意这种情况。

2. 使用三目运算符

在应用程序的大多数条件分支中我更喜欢用 if-else 结构。我觉得它在语句构成上比三目运算符更清晰。虽这样说，三目运算符也有适用的情况，例如，根据一个特殊值将另一个特殊值赋给一个变量的情况。代码清单 2-33 给出了一个例子。

代码清单 2-33　三目运算符

```
int result = (x > y ? 10 : 20);
```

三目运算符的作用和一个写在一行的微型 if-else 语句的作用相同。你可以将三目运算符看做由?和：分开的三个部分。?左边的部分是条件语句。如果条件语句为真，那么三目运算符的返回值是?和：之间的值，如果条件语句为假，三目运算符的返回值是冒号右边的值。

所以在代码清单 2-26 中，如果 x>y，结果将是 10，否则就是 20。正如你看到的，在赋值时使用三目运算符在两个值之间进行选择非常方便、简练。但简练也使三目运算符很难阅读，为了更易读我更倾向于使用 if-else 语句。不过，如果你觉得三目运算符可以使得代码更干净、清晰，尽一切办法使用它。它是工具箱中的另一个工具。

3. 使用 switch 语句

最后一种条件语句是 switch 语句。switch 语句是处理对不同选项进行分支的理想选择。你完全可以使用 if-else 语句代替 switch 语句，但是 switch 语句通常更清晰，并且有时候更快。

代码清单 2-34 给出了一个 switch 语句的示例。

代码清单 2-34 switch 语句

```
switch(state)
{
    case 1:
        doStateOneAction();
        break;
    case 2:
        doStateTwoAction();
        break;
    case 3:
        doStateThreeAction();
    default:
        doDefaultAction();
}
```

正如你所看到的，switch 语句是由 switch 关键字以及其后的圆括号中你想作为分支的值组成的。该值通常称为控制变量。在 switch 语句后，必须提供一个代码块，其中包含控制变量的可能值，以及这些可能值中的每一个要执行的指令。这些 case 分支的写法是 case 语句，跟着是一个冒号和对应的执行指令，接着是满足该条件时会执行的 break 语句。

break 语句是可选的，遇到 break 将停止执行 switch 并且程序的执行将跳出 switch 语句。如果没有提供 break 语句，后面的 case 语句将继续执行直到遇到 break 语句或者 switch 语句结束。

switch 语句中除了要提供 case 语句之外，还可以提供一个 default 部分。这在 switch 语句的末尾提供，在其他 case 都不满足的情况下执行。



说明

在代码清单 2-34 中，case 3 缺少 break 语句，所以如果进入 case 3，case 3 和 default 都将被执行。

值得注意的是 switch 语句的控制变量只能是整型数。不能使用字符串或者其他性质类似的控制变量来判断不同情况。另外，不能在 switch 语句内声明新的变量。

4. 条件语句的选择

因为 switch 的附加限制，它可以通过编译器优化，从而比 if-else 语句更高效、更快。不过，在大多数情况下，if-else 语句对于大多数你要执行的操作应该是足够快的。只有在极端情况下 switch 语句和 if-else 语句的性能差异才凸显出来。通常情况下，在两种条件语句中贸然优先选择某一种之前应该权衡一下性能。大多数情况下，你应该选择最能表达代码意图的条件语句。选择可以让后面的开发者（包括你自己）更易读的条件语句。

对我而言，较之 switch 语句我更喜欢 if-else 语句，因为我觉得它们更容易阅读。有的人会不同意，理由是长的 if-else 块很难理解。和大多数事情一样，你可能有不同的想法。

2.3.2 使用循环语句

循环可以让你的程序重复执行一系列指令直到满足某个条件。关于循环语句，你会遇到三种主要类型：for 循环、while 循环和 do-while 循环（它是 while 循环的变种）。

由于 Objective-C 2.0 及其快速枚举功能的出现，for 循环已经变成 Objective-C 中使用循环时的首选。不过，while 和 do-while 循环在语言中仍然起着重要的作用。

1. 使用 for 循环

for 循环可能是 Objective-C 中最常用的循环。你可以用它计算一定范围内的数，遍历一个数组中的项等。由于这种灵活性，它的语法中的一些变异可能让你感到困惑。

2. 传统的 for 循环

先来介绍 for 循环的传统形式。在这种形式中，for 循环语句由 for 语句和在圆括号内以 3 条语句形式存在的 for 循环的条件语句构成。这 3 条语句对应在 for 循环执行过程中对控制变量执行的 3 项操作。第一项操作设置控制变量的初始条件，它通常用来将变量初始化为 0，如果你不想从 0 开始计算时，可以将变量设置为其他值。第一项操作只在第一次进入 for 循环时执行。

第二项操作是条件语句，在每一次进入循环之前先判断循环是否应该终止。如果条件语句为假，for 循环中断并且程序执行 for 循环后的代码块。如果条件为真，再执行一次 for 循环。

最后一项操作是计数表达式，这个表达式用来在每一次循环中实际改变控制变量的值。每循环一次控制变量可以递增、递减、重新赋值等，这些都通过计数表达式实现。

代码清单 2-35 给出了传统 for 循环的例子。

代码清单 2-35　for 循环

```
for(int i = 0; i < 100; i++)
{
    Foo *foo = [array objectAtIndex:i];
    [foo doSomething];
}
```

在这种情况下，第一次进入 for 循环，创建控制变量（i）并将其初始化为 0。每循环一次，都会判断条件表达式以判断控制变量是否达到 100。如果达到，程序跳转到紧挨着 for 循环代码块的语句继续执行。如果没有，那么执行计数表达式。本例中，控制变量每次加 1。



说明

计数表达式可以是任何你想要的内容。例如，要想在循环中以 2 计数，只要在每一次循环中将控制变量递增 2 即可，而不是和这里一样递增 1。



说明

x++是后缀递增操作符。它增加 x 的值并将增加后的值存储到 x 中。这样做时，它返回的仍是 x 递增前的值。++x 是前缀递增操作符，它同样也增加 x 的值并将增加后的值存储到 x 中，但是返回的值是增加后的值。前缀/后缀递减操作符的工作原理与此类似。某些情况下，你要将这些操作的返回值赋给其他变量，执行此操作时要清楚这一特性。

控制变量在 for 循环代码块作用域内有效并且可在 for 循环内使用，例如，作为数组的索引，或者改变 for 循环内部的条件逻辑。此外，执行 continue 语句可以在 for 循环代码块内的任意位置中止 for 循环。continue 语句可以马上停止 for 循环的执行并且使执行回到循环的开始。代码清单 2-36 展示这个例子。

代码清单 2-36　continue 语句

```
for(int n = 0; n < 100; ++n)
{
    if(n > 10 && n < 20)
        continue;  //跳过 11 到 19
    //对 n 进行操作
    Foo *foo = [array objectAtIndex:i];
    [foo doSomething];
}
```

3. 使用 for 循环进行快速枚举

第二种形式的 for 循环是 Objective-C 2.0 的新增特性。它适用于枚举集合中的对象。你在下一章才会学习对象和集合，但是我想在这里介绍一下快速枚举语法，这样到时候你就不会感觉太陌生。

在前面那种形式的 for 循环中，我们需要使用控制变量来获取数组的元素以便对它进行操作。快速枚举形式的 for 循环废弃了这种机制，取而代之的是可以在 for 语句中指定一个临时变量来存储集合中被遍历的元素。这是非常方便的，因为我们写的大多数的 for 循环就是用来遍历集合中的成员并对每一个成员单独的执行一个操作。

代码清单 2-37 展示了一个快速枚举的例子。

正如你所看到的，一个变量（对象）声明为 for 语句的一部分。执行 for 循环时，集合（数组）的每一个成员都被赋值到那个变量。那么这个变量在 for 循环代码块的作用域内有效。

代码清单 2-37　快速枚举的 for 循环

```
for(Foo *object in array)
{
    //Foo 对象被赋了一个数组成员的值
    //操作对象
}
```

在第 13 章讲到集合对象时你会看到使用多个 for 循环的例子。现在只要简单熟悉这种形式的 for 循环，需要的时候可以随时查阅本章。

4. 使用 while

while 循环除了在 while 语句后的圆括号内没有多个语句外，工作原理和 for 循环类似。while 语句有一个每一次循环都检查的条件语句。当条件语句为假时，程序继续执行 while 循环代码块后的语句。

代码清单 2-38 展示了一个 while 语句的例子

代码清单 2-38 while 循环

```
int x = 0
while(x < 10)
{
    NSLog(@"Value of x: %ld", x);
    x++;
}
```

使用这种控制流应该注意的是在某种情况下，while 代码块要将 while 条件语句中所检查的条件变成 false。如果没有这样做，while 语句将会是死循环。和 for 循环一样，while 循环也可以用 continue 语句中断[①]。



说明

如果条件语句在第一次执行循环时为假，while 中的代码块将不会被执行。

while 语句在涉及复杂的遍历逻辑时非常方便。

早期版本的 Objective-C 不支持现在的快速枚举。于是，经常用 while 语句通过 NSEnumerator 来遍历数组。这样的例子如代码清单 2-39 所示。

代码清单 2-39 旧式遍历

```
NSArray *someArray = [self getArray];
NSEnumerator *enumerator = [someArray objectEnumerator];
while((NSObject *obj = [enumerator nextObject]))
{
    //对 obj 进行操作
}
```

现在你很少看到这样的模式了。但是因为它在早期的代码中普遍存在，熟悉这个概念很重要。每一次检查条件语句时，枚举器返回数组的下一个对象。到达数组的结尾时，它返回 nil。这导致 while 循环条件返回假。这又使程序跳出 while 代码块并执行它后面的代码。

5. 使用 do

本节将要介绍的最后一种循环类型是 do-while 循环。这种循环除了将条件移到了循环的结尾以外，其他地方和 while 循环类似，因此在检查条件前至少会执行一次循环。该类型循环的示例如代码清单 2-40 所示。

这种循环很少使用，但是使用时，它通常是你要尝试解决的问题的一个很好的解决方案。

① 原文使用中断，这里说明一下：continue 会致使循环跳过循环体中余下的语句，转而判断循环条件是否仍然成立，然后选择是否再次进入循环体；break 的作用是彻底跳出当前循环而执行循环体后的语句。——译者注

代码清单 2-40　do-while 循环

```
int x = 0;
do
{
    NSLog(@"%ld", x);
    x++;
}
while(x < 10)
```

通常，如果你在处理一些已经有状态值的控制变量，并且不想改变它，直到循环至少执行过一次，这种情况可以使用这种循环。当你在控制块的作用域内计算控制变量时也可能用到这种循环。同样，为了得到控制变量的值你可能要执行一些复杂的逻辑。`do-while` 循环在判断条件前执行你在循环体内设计的复杂逻辑。但是注意，控制变量必须在 `do-while` 代码块外部声明，这样它才能在 `while` 条件的作用域内有效。



说明

记住，在处理 `switch` 时，使用 `break` 语句可以跳出 `switch` 语句。这在 `switch` 的使用中非常常见。然而，`break` 同样可以在 `do-while`、`while` 和 `for` 循环中使用。在这些情况下使用时，它会使执行线程跳出循环，跳到程序中循环体后面的一行。

2.4　活学活用

现在你可以创建一个命令行计算器程序了。虽然你可能不能理解这个程序的全部内容，但是应该理解了大部分内容，下一节将解释你还没有理解的部分。

这个程序通过命令行获取一系列参数，这些参数是以运算符隔开的数字。因此，你可以这样运行程序：`./Calculator '10 + 5 - 3'`。程序按顺序加、减、乘或除这些数值，完成时，它在控制台输出最终结果。为了演示，我将程序分成两个单元：包含了主要功能逻辑的主单元和包含加减等操作函数的 `MathOperations` 单元。要开始程序，打开 Xcode 并创建一个新的基于 Foundation 的命令行程序，并将其命名成 Calculator。



说明

不用担心 `NSString`、`NSArray` 等调用，这些将会在下一章介绍。项目创建后，打开 Calculator.m 文件，并按照代码清单 2-41 进行修改。

代码清单 2-41　Calculator.m

```
#import <Foundation/Foundation.h>
#import "MathOperations.h"

BOOL isAnOperator(const char value)
{
```

```
    return ((value == '+') || (value == '-') || (value == '*') || (value ==
   '/'));
}

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    double result = 0;

    char operator = '\0';

    NSString *equation = [NSString stringWithUTF8String:argv[1]];
    NSArray *eqParts = [equation
                        componentsSeparatedByCharactersInSet:
                        [NSCharacterSet whitespaceCharacterSet]];
    for(int n = 0; n < [eqParts count]; n++)
    {
        NSString *argString = [eqParts objectAtIndex:n];
        char firstChar = [argString characterAtIndex:0];

        if(isAnOperator(firstChar))
        {
            operator = firstChar;
            continue;
        }

        double newValue = [argString doubleValue];

        switch (operator)
        {
            case '+':
                result = add(result, newValue);
                break;
            case '-':
                result = subtract(result, newValue);
                break;
            case '*':
                result = multiply(result, newValue);
                break;
            case '/':
                result = divide(result, newValue);
                break;
            default:
                result = add(result, newValue);
                break;
        }
    }

    NSLog(@"%.3f", result);

    [pool drain];
    return 0;
}
```



说明

前面我说过 switch 语句只能使用整数，可是，这里我使用了 char 型，回顾一下，我在介绍标量时提到 char 型在内部实际是用整型来表示的。使用单引号''，编译器自动将单引号内的字符值转换成代表这个字符值的整数。

下一步，创建一个 C 源文件，将其命名为 MathOperations.m，并使其和代码清单 2-42 一样。

代码清单 2-42　MathOperations.m

```
#include "MathOperations.h"

double add(double value1, double value2)
{
    return value1 + value2;
}

double subtract(double value1, double value2)
{
    return value1 - value2;
}

double multiply(double value1, double value2)
{
    return value1 * value2;
}

double divide(double value1, double value2)
{
    return value1 / value2;
}
```

最后，按照代码清单 2-43 编辑 MathOperations.h。

代码清单 2-43　MathOperations.h

```
extern double add(double value1, double value2);
extern double subtract(double value1, double value2);
extern double multiply(double value1, double value2);
extern double divide(double value1, double value2);
```

之后，编译应用并打开终端程序。你应该能够在项目目录下 build/Debug 子目录中找到编译过的程序。如图 2-8 所示进入这个子目录，并运行你的新程序。确保在你运行它时输入的单引号和我这里给出的单引号一样，这会让控制台将'*'字符作为通配符忽略并传递给你的程序。

这个程序结合了前两章的所有知识：for 循环、条件语句、函数、基本类型等。



图 2-8　程序的输出

2.5 小结

本章介绍了 Objective-C 中面向过程程序设计的全部基本语法。通过本章你将学会如何声明变量和结构体，如何使用运算符，以及如何使用函数。你还学会了使用循环和条件表达式在运行时控制程序流。最后，结合这些知识做一个简单的计算器程序。

下一章将深入研究 Objective-C 的对象部分并看看 Objective-C 如何实现面向对象编程。

第 3 章

添加对象

3

本章概要

- ❑ 学习面向对象术语
- ❑ 在 Objective-C 中使用对象
- ❑ 创建类和类层次结构
- ❑ 定义属性
- ❑ 编写类和对象方法
- ❑ 声明成员变量

从根本上讲，所有软件开发技术的最终目标都是为了解决一个问题。这个问题就是：人类很难同时考虑多个想法。因此，利用所有这些技术我们可以将我们的想法分块和封装成可以复用的包，并且能够通过这些包的新组合和匹配方式来解决新问题。

我们已经了解了过程化编程如何将想法分解成可以复用的过程，从而实现分块和封装。过程编程在最初引入到计算机科学时曾是一个革命性的概念。但是，它有一个很大的缺陷：过程没有可以存储状态的机制。面向过程编程的程序员一般通过参数来传递变量到函数或者依赖全局变量来存储状态来避开这样的限制。但这种方法都不是理想的解决方案。

随着程序变得越来越复杂，需要在过程调用之间保存越来越多的状态。这样，向过程传递状态变量很快就会变得不可控。

使用全局变量也同样复杂，因为全局变量的过度使用会导致代码中难以跟踪的纷繁复杂的依赖关系。比如，为了确定某个过程依赖于哪些全局变量就必须熟悉过程本身。没有一种方法仅从过程接口就可以看出该过程所依赖的全局变量。这就会导致变量被不正确地初始化，或者函数中接收值的变量在没有觉察的情况下被访问等副作用。

随着过程化应用程序变得越来越复杂，对程序员来说，这些问题变得越来越难以克服。

因此需要一种新的编程方法，它支持程序员将用于操作的数据以及操作这些数据的逻辑一起封装到一个包里。这种新的编程方法就是面向对象编程。接下来的几节会介绍该技术。

3.1 对象

面向对象编程背后的思想就是支持程序员将要操作的数据以及操作这些数据所需的过程封

装起来。

比如，试想下你需要封装一只猫的行为。所需要做的第一件事就是记录猫的特性（attribute）——它的数据。图 3-1 展示了这种概念。你可能想记录猫的颜色属性，在本例中，猫的颜色是黑色。

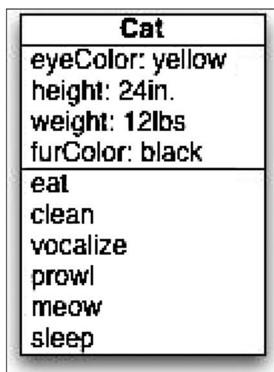


图 3-1　封装猫的属性

你可能还想把猫的眼睛是黄色这一实际情况记录下来。有些猫的眼睛是灰色的，而有些则是绿色的，但这只猫的眼睛是黄色的。可能你还想记录猫的重量。所有的这些都是猫的特性。如果你想在一个过程式编程的应用中处理这些，所有这些属性都必须存储在某种类型的全局变量中。通过使用全局变量的结构体可能会稍微缓解一下这种问题，但即使如此，在需要更多的结构体时，管理一堆结构体很快就会变得不可控了。

通过面向对象编程，你可以使用对象来表示猫的状态和特性。同样地，猫的一个特定实例是一个更通用、更理想的“猫”概念的实际表现。这个概念可以说是名为猫的很多猫的一个通用版本。这就称作对象的“类”。对象的类的另一种解读就是它是猫这一概念的一种纯理论的表现。猫的纯理论表现是一种概念模板，你可以用它来创建每一只真实的猫的实例。例如，一只猫（名字叫 George）是猫“类”的“对象”或者“实例”。

从另一个角度看，`Cat` 类代表了构成猫的所有特性的定义，这包括不同种类的猫可能具备的所有可能的类型。使用这个模板，你就可以创建一个单独的实例来描述特定的猫。`Cat` 类可能会定义出一个毛茸茸的、耳朵尖尖、牙齿锋利的以及有攻击主人的腿倾向的猫。该类还可能定义一个有不同颜色的眼睛或者皮毛的猫。你还可以把猫的某个特定实例的皮毛颜色定义成一个变量，用来区分具体的猫和通用概念上的猫。

除了定义猫的通用概念的特性以及某一只猫的具体特性外，类还可以封装猫的行为。比如，猫是喵喵地叫。可以在 `Cat` 模板中创建可以让猫喵喵叫的代码，这就叫做方法。同样，你还可以在猫类中定义一个表示自己清洁的过程的方法。猫可能有一个表示“脏”的临时状态。“清洁”方法就表示将脏猫变成干净猫所需的行为。这样，猫类中定义的方法就定义了一个改变猫的特定实例中的数据所需的指令。

撇开这个比喻，我们回到实际的 Objective-C 中。在这里需要知道的是 Objective-C 定义了一个可以定义这些概念和关系的编程结构。

在 Objective-C 中类表示特定类型对象的定义或者模板。当使用对象时，你需要创建类。在这些类中需要定义对象所封装的数据以及用于操纵这些数据的方法。

对象是一个类的特定实例。对象通常也称作实例，这两个词是可以互换的。

封装在一个对象中的数据可以称作数据、状态、特性或者属性。但是，需要注意的是还有另外一个称作属性的编程概念，本章稍后将会介绍。数据不一定是属性，反之也成立。

最后，当提到一个对象的行为，包括改变其数据的行为时，和该行为相关的计算机指令称为方法。Objective-C 开发者倾向于使用 Smalltalk 中的惯例，称方法为消息。该术语在将方法作为一项操作使用时最常用。一些编程语言将这称为“调用一个方法”，但 Objective-C 程序员更倾向于使用“发送一条消息”，我则是更喜欢使用与“方法”相关的术语，但是有时也会使用消息这一术语。



说明

在学习本章余下部分之前，理解用于描述类、对象、属性和方法的术语很重要。如果有任何疑惑，请重新阅读 3.1 节以清楚地了解这些术语之间的差别。

1. 继承

继承有两个方面。一方面表示类继承的类设计，这会影响到如何设计类以及在类定义的何处声明行为。类继承的另一方面与外部视角中类的样子以及如何使用类相关。（这一主题会在“多态”中介绍）

让我们再来看看 `Cat` 类，回忆一下代表宠物模板的 `Cat` 类。你可以将其视为宠物的物种表示。跟其他物种一样，`Cat` 类可以说是从其他物种发展而来的。换句话说，猫属于哺乳动物，并继承了哺乳动物的部分特性。同样，哺乳动物从脊椎动物发展而来并继承了它的特性。猫还有“姐妹”物种，比如狗。每个物种都和其他物种以及父类物种有共同的特征。你可以在任意方向上扩展，使之变得更具体（贵宾狗、德国牧羊犬等）；或者变得更通用，顺着继承树到哺乳动物、脊椎动物等。图 3-2 通过排布几类不同生物以及它们之间的联系大体展示了这种概念[①]。

Objective-C 中的类有同样的能力。事实上，像物种继承树一样，图 3-2 也可以很容易地表示类的继承树。换句话说，Objective-C 中的类可以有父类，这些类继承了父类的行为和特性。此外，你创建任何的类都可以有继承该类行为和属性的子类。

① 图 3-2 并没有体现物种数的关系。——译者注

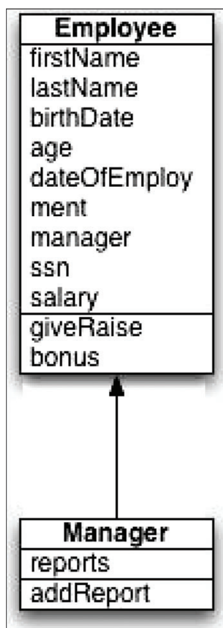


图 3-2 类继承树

在本例中，`Cat` 类继承了 `Mammal` 类。`Mammal` 类指定所有子类都有皮毛、能生崽、能吃、能睡、能发声等。结果，它给出了皮毛颜色、眼睛颜色等特征。同时也给出了吃饭、睡觉、发声等的标准方法。哺乳动物的子类可以自动继承这些特性和行为，在需要的时候时可以重写它们。比如，猫的独有发声和狗的发声不一样。猫的发声是喵，狗的发声是吠。在新子类的实现中为了重写父类的行为，你可以直接在子类中定义该方法。新的版本会重写父类中的版本，并会在调用该方法时由运行时自动使用。

在面向对象的语言中定义类时都可以使用这样的可定制功能。在 Objective-C 中，一个指定类可以从任何其他指定类继承。但是，一个类只能有一个父类。类本身会继承父类及父类的父类的所有特性和行为，但是无法从多个的直接父类继承行为。这种概念就是单父类继承。C++等语言支持一个类从多个父类继承，但这会导致编译时的二义性问题。结果，Objective-C 选择了只支持单父类继承来避免这样的二义性。物种继承树的树根是一个祖先类，它是所有生物的祖先，Objective-C 中也有这样一个类，所有的类都继承于它，该类就是 `NSObject`。`NSObject` 提供了 Objective-C 中所有类都需要的最基本的功能，如内存管理例程、复制例程等。



说明

在 Objective-C 中，可以在不使用多个父类的情况下模拟多父类继承。在介绍协议和类别时我会介绍如何实现。

2. 多态

本章前面介绍了类继承的概念。多态是指从一个给定类创建的任何对象都可以被该对象的使用者视为是该类的实例，或者是该类的父类的实例。这就意味着你的代码可以利用它所继承的类的最基础的功能。由于 Objective 中的类最终都继承自 `NSObject`，你甚至可以创建一个只需要 `NSObject` 就可以运作的代码。这段代码不一定很有用，但是会很灵活，因为你可以将任何对象作为参数传递给它，它能正常运行。

继续使用之前的动物比喻，你可以试想着编写要求哺乳动物对象发声的代码。但是现在使用的哺乳动物的特定实例其发声方式可能会有很大不同，在你调用发声方法时，它会判断你处理的是什么类型的哺乳动物，然后调用相应的方法。比如，狗会吠，猫会喵，狮子会吼。

这是一个极其强大的功能，在你的编程生涯中你会不断用到它。多态赋予了面向对象编程真正的威力。

3. `id` 数据类型

在分析本章中的代码之前我想再介绍一个概念。这一概念就是 `id` 数据类型。`id` 数据类型是一种能在 Objective-C 中表示所有对象的特殊数据类型。它可以用在所有使用对象类型的场合。

大多数情况下，不需要直接在代码中使用 `id` 数据类型。但是，你可能会遇到一些在 Cocoa 和 Cocoa Touch 库中使用 `id` 数据类型的情形，尤其是用在数组、字典中。在这些情形下，`id` 数据类型的使用及其可以“伪装”成任何类的能力实际上赋予了 Objective-C 动态类型的能力。

你可能会暗自发问：“为什么不都使用 `id` 呢，这样就不用在声明实际对象类型时劳心了？”遗憾的是，由于在使用 `id` 数据类型而不是更具体的类型名的情况下，编译器在编译时不知道你所使用的对象类型，这就可能放过一些原本可以被编译器捕捉的错误。此外，运行时查找数据类型需要额外的开销，因此声明为 `id` 数据类型的对象的方法调用会比更具体数据类型的对象的方法调用稍微慢些。因此，一般来说，最好将对象声明成具体类型，而不是使用 `id` 数据类型。不过了解 `id` 数据类型的概念是很重要的，这样就可以在需要的特殊场合下使用了。



说明

在初始化方法中使用 `id` 数据类型。

3.1.1　创建类

到目前为止，我介绍了面向对象编程的总体概念：类、继承和多态。在接下来的几节中，我会介绍如何实际创建类并在代码中使用类。

1. 使用类文件

类通过两个单独的文件定义。第一个文件是接口文件。接口文件的扩展名为.h。在该文件中你可以定义类的接口。代码清单 3-1 给出了一个例子。

代码清单 3-1 接口文件

```
#import <Foundation/Foundation.h>

@interface Foo : NSObject
{
    NSString *someVariable;
    NSString *someOtherVariable;
    NSArray *someArray;
}
@property (nonatomic, retain) NSString *someVariable;
@property (nonatomic, retain) NSString *someOtherVariable;

-(void)someMethod;
-(BOOL)someOtherMethodWithArg:(NSString *)param andAnotherArg:(int)param2;

@end
```

可以看出，类接口通过`@interface Classname`这种特殊语法声明。`Classname`表示要定义的类的名字。如果类从其他类继承，可以在类名后列出将要继承的类名。本例就简单地从`NSObject`继承。

在`@interface`的下一行是一个用大括号包围的代码段。在大括号中可以定义类数据。在大括号中定义的变量的作用域就是作为类的一部分定义的任何方法。

旧的运行时环境要求所有的成员变量都必须在此定义，但是，现在的 64 位运行时环境已经没有这个限制了，Mac OS X 10.6 和 iOS 都已经可以支持 64 位运行时环境了。你可以简单地将成员变量作为属性进行声明。在此额外定义这些成员变量也无妨，因此有这样的习惯也没有什么错。

在大括号后就可以定义作为类一部分的方法签名。最后，你可以通过`@end`指令标志该接口结束。

在创建了类的接口定义后，还必须实现它。类的实现在一个以.m 为扩展名的文件中创建。我们刚看到的类的示例实现如代码清单 3-2 所示。

代码清单 3-2 示例实现文件

```
#import "Foo.h"

@implementation
@synthesize someVariable;
@synthesize someOtherVariable;

-(void)someMethod
{
    //方法体
}

-(BOOL)someOtherMethodWithArg:(NSString *)param andAnotherArg:(int)param2
{
    //方法体
}

@end
```

同样，我们有个特殊的语法来告诉编译器：我们要创建一个实现。语法就是@implementation 指令。和@interface 类似，需要将要定义的类名放在它后面。在@implementation 和@end 指令之间就可以定义所有在类中使用的方法。

回到接口文件，让我们深入学习一下如何在类中封装数据。

2. 编写对象方法

对象方法是作为类的一部分定义的，是只有在对象实例化后才能调用的方法。通常，这些方法就是在介绍用于操作对象内部数据的方法时提到的方法。如用于改变对象内的数据或者基于对象内的数据进行计算的方法通常都是以对象方法的形式实现的。

创建一个对象方法包括两个部分。第一部分是在类接口文件中声明的方法签名。代码清单 3-3 给出了一个例子。

代码清单 3-3　对象方法声明

```
-(BOOL)someOtherMethodWithArg:(NSString *)param1
                andAnotherArg:(int)param2
```

所有的对象方法都以连字符（-）打头，从而同以加号（+）打头的类方法区分开来。



说明

类方法是可以使用未实例化的类而不是对象调用的方法。

方法的返回值类型在小括号中指定。返回类型后是方法名以及方法的参数。每一个参数在冒号后指定。参数的类型在小括号内指定，然后指定名字。最后，方法的结尾必须有一个分号。

在声明了要创建的方法的方法签名后，你需要创建实际的实现。这也称为*方法定义*。方法定义在类的实现文件中。在创建方法实现时，方法实现的第一行必须和接口文件中的方法接口声明一致，然后在大括号中放置方法体。代码清单 3-4 展示了我们前面声明的方法的实现。

代码清单 3-4　对象方法定义

```
-(BOOL)someOtherMethodWithArg:(NSString *)param1
                andAnotherArg:(int)param2
{
    //这里利用 param1 和 param2 实现一些东西
    if([someOtherObject doSomething:param1] == param2)
        return YES;

    return NO;
}
```

方法的实现指定了在执行方法调用时要求计算机执行任何行为所需的指令。若要从方法返回值，需要像前面定义过程一样使用 return 语句。

类的所有数据成员都在该类的对象方法的范围内可用。此外，所有作为参数传入到该方法的变量也在该函数内可用。

3. 使用特殊对象方法

除了按照你的功能性需求来定义行为的对象方法外，作为类的一部分，你还可以选择性地定义一些特殊的对象方法。这些方法具有特定的功能以及标准的行为。这样的方法有很多，目前我只想介绍其中两个。

3

这些特殊对象方法中的第一个就是一系列的初始化函数。初始化函数方法总是以 `init` 打头并返回 `id` 数据类型。除了这些惯例外，这些方法的方法签名是很随意的。但是，初始化方法的方法体应该遵循一种特殊的标准化语法。一个典型的初始化函数的示例如代码清单 3-5 所示。

代码清单 3-5　典型的初始化函数

```
-(id)init
{
    if((self = [super init]))
    {
        memberVariable = [[NSMutableArray alloc] init];
    }
    return self;
}
```

初始化函数方法的结构很重要。第一步就是调用指定的父类初始化函数。该初始化函数返回父类对象的一个经过初始化的实例，并且必须将其赋给特殊的变量 `self`。如果在初始化过程中出现任何问题，初始化函数的协议指定返回一个 `nil` 对象，而不是一个有效的初始化对象。因此，在将父类的初始化函数返回值赋给 `self` 后，必须检查 `self` 是否是 `nil`。如果是，就不想要初始化自身的变量了，返回 `nil` 即可。在上面给出的示例中，我们在 `if` 语句中将变量赋给 `self` 并检查它是否是 `nil`。

初始化函数的真正目的除了创建 `self` 外，还用于初始化对象中的所有数据成员。因此在验证 `self` 不是 `nil` 后，就可以初始化变量了。在初始化变量后，就可以从初始化方法中返回 `self`。

在某些情况下，在一个类中提供多个初始化方法比较合理。比如，如果有不同方式可以创建对象并且在这些状态下需要传入不同的参数。在这些情况下，需要创建多个不同的初始化函数来接收不同的参数。为了避免重复的代码，可以在初始化函数中调用其他的初始化函数。这样就可以保证初始化仅仅在一个地方进行。

除了初始化函数这种概念外，还有指定初始化函数的概念。通常它的参数个数是所有自定义初始化函数参数个数的最小值，并且是其他所有初始化函数在设置对象的初始状态时最终调用的初始化函数。

该示例如代码清单 3-6 所示。

代码清单 3-6　不同的初始化函数调用指定初始化函数的示例

```
-(id)init
{
    if((self = [super init]))
    {
        memberVariable = [[NSMutableArray alloc] init];
    }
    return self;
}

-(id)initWithArray:(NSMutableArray *)inArray
{
    if((self = [self init])) ①
    {
        memberVariable = [inArray retain];
    }
    return self;
}
```

第二个必须熟悉的特殊方法是 `dealloc` 方法。`dealloc` 方法和 `init` 方法是相对的。它用于释放在 `init` 方法或者其他地方分配的资源。你必须在该方法退出前调用父类的 `dealloc` 方法。`dealloc` 方法示例如代码清单 3-7 所示。

代码清单 3-7　`dealloc` 方法示例

```
-(void)dealloc
{
    [memberVariable release]; memberVariable = nil;
    [super dealloc];
}
```



警告

在释放自定义变量之前不能调用父类的 `dealloc` 方法，这会导致程序崩溃。

第 4 章会更详细地介绍初始化函数和 `dealloc` 方法。目前重点是要熟悉这些方法，因为在接下来的示例代码中会使用这些方法。

4. 编写类方法

在 Objective-C 中，类本身可以有很多和对象一样的功能。比如，类可以声明直接通过类本身调用的静态方法。在使用这类方法时不需要实例化所使用的类的实例。这对于用于创建一些类（例如工厂类和单例类）的实例的方法来说很方便。实际上，Cocoa 框架利用很多内置类中的方法创建类的实例。在其他语言中，比如 C++或者 Java，类方法通常称为“静态方法”。如果你很熟悉这些语言的话，那么你应该也很熟悉这个术语。

① 此处会导致内存泄漏。——译者注



说明

Objective-C 的工厂方法就是为方便创建对象而使用的类方法。它总是返回一个自动释放的对象。第 4 章会详细介绍工厂方法。

3



前后参照

第 17 章将介绍单例。

声明一个类方法和声明一个对象方法很类似，唯一的不同就是在方法声明前不是使用连字符，而是使用加号。类方法声明的示例如代码清单 3-8 所示。

代码清单 3-8　类方法声明

```
@interface Foo : NSObject
{
    NSMutableArray *memberVariable;
    NSString *anotherMemberVariable;
}
@property (nonatomic, retain) NSMutableArray * memberVariable;
@property (nonatomic, retain) NSString * anotherMemberVariable;

-(id)init;
-(id)initWithArray:(NSMutableArray *)inArray;

//一个类方法
+(id)fooWithArray:(NSMutableArray *)inArray;

@end
```

类方法的实现和对象方法的实现一样。但是，类方法不能访问对象的成员变量。至于为什么，记住，类方法是直接从类本身调用的，而不是从类的实例调用。因此，没有对象用来储存这些用于操作的数据。类方法的实现示例如代码清单 3-9 所示。

代码清单 3-9　类方法定义

```
@implementation Foo

+(id)fooWithArray:(NSMutableArray *)inArray
{
    return [[[self alloc] initWithArray:inArray] autorelease];
}

@end
```

和本例中的方法一样，这种类方法最常见的应用就是在工厂类中使用。Cocoa 和 Cocoa Touch

中的很多类都有工厂方法，它们使得对象创建变得更容易。比如 NSArray 就包括一个工厂方法 [NSArray array]来返回经过恰当构造的 NSArray 对象。（第 4 章将介绍一些特殊的内存管理规则）。在创建类方法时可以按照本示例所示使用 self 对象来指代类本身。

3.1.2　声明对象

之前已经介绍了如何声明类，包括数据、方法等。但如果不能创建类的实例并使用它们，那么所有这些就变得毫无意义。现在我们就看看如何在代码中声明一个类的实例。

代码清单 3-10 显示了利用上一节所创建的类进行一些有用操作的代码示例。

代码清单 3-10　创建一个自定义类的实例

```
{
    //旧式的初始化函数
    Foo *object;
    object = [[Foo alloc] init];
    [object doSomethingWithParameter:arg];

    //所有都在一行
    Foo *object = [[Foo alloc] initWithArray:[NSMutablArray array]];
    [object doSomethingWithParameter:arg];

    //使用类工厂方法
    Foo *object = [Foo fooWithArray:[NSMutableArray array]];
    [object doSomethingWithParameter:arg];
}
```

可以看出声明一个自定义类的实例很简单。首先，使用带有指针操作符（*）的类名。然后是用于保存类实例的变量名，在本例中是 object。你可以选择在变量声明的同一行立刻初始化变量。为此，只需使用等号操作符来赋值就可以了。在代码清单 3-10 的示例代码中使用了两种方式。第一种是不初始化变量，另一种就是在一行内初始化变量。在该代码中有一种重要的新语法。在本书中前面的其他部分你已经见过这种语法，接下来我详细介绍一下它。这里所说的语法就是中括号操作符（[]）的使用。

在 Objective-C 中，对类和对象上调用方法时要将它们都放在中括号中。你可以在与方法相关的对象或类之前放置左边中括号，在方法调用末尾放置右边中括号。所以，在上面的代码中你可以看到在对象变量初始化的过程中，我们实际上在两段初始化代码中调用了两个方法。第一个是调用 alloc。该方法实际是在类上调用。第二个是调用 init。该方法实际是在 alloc 调用所返回的对象上调用的。重点是要知道 alloc 调用实际上会返回一个对象，而 init 方法就是在返回的对象上调用的。在 Objective-C 中这种在前一个方法调用返回的对象上调用方法的嵌套调用并不少见。

初始化 Objective-C 对象实际上分两步。第一步就是分配内存用于存储构成对象所需的数据和方法。这就是调用 alloc 的目的。这也是可以在 alloc 方法返回的对象上调用 init 方法的原因。

第 4 章会更详细地介绍这部分内容，要点就是任何通过 alloc 方法分配的对象（如本例所示）都必须被释放。要释放对象，你可以简单地在对象上调用 release 方法。release 是一个

在 NSObject 中定义的方法。

3.1.3 调用对象方法

在声明并创建了一个自定义类的实例后，你可能就需要在该实例上调用方法了。可以使用上一节介绍的中括号语法调用在类中声明的任何对象方法。但是只有在类定义的接口文件中声明的方法才能从该类以外的模块调用。尽管从技术角度来看，调用没有在接口文件中声明的方法是可行的，但编译器还是会在编译代码时发出一个警报。此外，编译器无法确定这类方法需要的参数类型和返回值类型，因此就无法捕捉一些正常情况下可以捕捉到的错误。

在某些情况下，你可能想在类中创建一些只在类的内部使用的方法。也就是你不想把它们的功能公开给外部的类。在 C++或者 Java 等其他语言中，你可以将这些方法看做“私有”方法。Objective-C 没有声明私有方法的语法，但是如果不在接口文件中公开方法，其他类如果调用了这些方法就认为是一种糟糕的做法。部分原因在上一段已经说明，还是部分原因是惯例。没必要将仅仅在类中使用的方法公开给该类的使用者。你可以单独在实现文件中声明类方法，这样就可以在实现文件中使用。要点就是在使用方法前编译器仍然需要知道方法签名。因此，仅仅在实现文件中使用而不想对外公开的方法应该放置在调用该方法的方法之前。示例如代码清单 3-11 所示。

代码清单 3-11　Objective-C 中的“私有”方法

```
@interface Foo : NSObject
{

}

+(id)fooWithArray:(NSMutableArray *)inArray;
-(void)someOtherMethod;

@end

@implementation Foo

+(id)fooWithArray:(NSMutableArray *)inArray
{
    return [[[self alloc] initWithArray:inArray] autorelease];
}

-(void)somePrivateMethod; ①
{
    //这里是私有函数的实现
}

-(void)someOtherMethod;
{
    [self somePrivateMethod] //没有问题
```

① 注意，在实现文件中的函数结尾不应该有分号，此处原文有误，下文相同。——译者注

```
    [self anotherPrivateMethod];  //这儿会有一个警告
}

-(void)anotherPrivateMethod;
{
    //这里是私有函数的实现
}

@end
```

注意，在该代码清单以及文件中第一个私有方法在调用它的方法之上，第二个私有方法在调用它的方法之下。如果你也打算这样处理代码，调用第二个私有方法时会产生一个编译器警告，这和调用一个其他类的私有方法类似。



说明

由于和文件中方法声明的位置相依赖，你可能会觉得这种“私有”方法的声明方式太粗略了。通过类别声明私有方法的方式可以参考第 8 章。

3.2　使用属性

Objective-C 最近新增的一个概念就是属性。属性可以用于声明类的数据成员的存取器方法。有了它们就不必使用很多之前为了访问数据成员所需要的标准代码（boilerplate code）。这也使得类的开发者可以定义对象状态的协议。这是 Objective-C 一个重要的新增语法。

3.2.1　状态和行为的区别

上一节提到对象是封装特性和行为的主要机制。本节会更深入地讨论该主题以解释一些有助于使用 Objective-C 属性的概念。

Objective-C 中的属性可以帮助你公开代表对象状态的对象属性。在内部实现上属性被编译成可以用于获取或设定对象数据的实际方法。这些方法称为存取器函数。你可以选择编译器自动生成的存取器函数，或者自己编写来重写编译器的存取器方法。



说明

存取器函数是专门用来供对象的使用者设置或获取对象中值的一个方法。它们封装了对象的数据成员并对外隐藏了对象中的实现细节。存取器函数可以直接访问变量，或则在访问时进行一些计算。有时，这些方法也称为赋值函数（setter）和取值函数（getter）。大多数人都会使用属性，而不会手动编写存取器函数，但在想重写属性存取器函数的常见行为的情况下，可以提供一个自定义的实现轻松重写存取器函数。在介绍属性的那一节就会涉及这方面的内容。

对象由状态和行为组成。状态包括构成对象的数据。在考虑对象状态时，大多数开发人员遵循的一个好的设计规则：虽然可以在任何时候改变对象的状态，但状态设定后直至应用程序改变它为止一直保持状态是比较安全的。改变一个对象的状态却造成副作用是一种不好的做法。如果需要进行一个会导致副作用的对象状态改变，就需要认真考虑造成这种结果的设计决策。

另一方面，行为可以看作是对象执行的操作。行为可以用于更新其他对象，因此也有副作用，或者也可以用于改变对象的内部数据或则触发对象上的其他操作。

面向对象设计的一些学派宣称对象必须仅向外部实体公开行为。利用 Objective-C 2.0 的属性标记，Objective-C 2.0 的设计者觉得公开对象的内部状态也是可行的。利用属性标记开发者在公开对象状态的同时也提供了访问状态必须使用的存取器函数。同样地，属性只能用于访问和控制对象状态。更清楚点说，在编写属性时，你不能创建一个具有有大范围外部影响的属性。这样的任务只能由对象的行为来实现。

举个例子，假设有一个代表引擎的类。该引擎类可能公开一个指定油门的属性。指定油门的属性就只能设定油门大小。如果想额外公开一种行为，使得引擎根据油门大小来改变行为，那么你需要添加一个 `updateEngineSpeedFromThrottle` 之类的方法（行为）。该方法没有输入参数并返回一个表明是否根据油门大小成功更新引擎速度的布尔值。

通过分离类的状态和行为，你可以避免潜在的副作用以及行为和对象特性之间的依赖关系。

1. 利用属性声明对象状态

对于这方面的例子，我们可以假想一个人力资源（HR）的应用。该应用可以用来跟踪员工福利、包括工资、保险等。因此我们需要创建一个用于封装这些数据的 `Employee`（员工）类。

我确信你可以想象到员工类需要封装的不同特性和属性。通常这会包括员工名字、姓、社保号、员工号、工资、可能还有经理等其他员工的引用。所有这些项都可以利用员工类的属性来表示。

代码清单 3-12 展示了如何创建员工类接口。它包括我刚才指定的数据成员列表以及访问这些数据成员的属性。在我们逐一介绍这些属性的时候，你就会理解每个属性的工作原理以及这些属性具备哪些特性。

代码清单 3-12 员工类接口

```
#import <Cocoa/Cocoa.h>


@interface Employee : NSObject
{
    NSString *firstName;
    NSString *lastName;
    NSDate *birthDate;
    NSDate *dateOfEmployment;
    Employee *manager;
    NSString *ssn;

    double salary;
```

```
}
@property (nonatomic, retain) NSString * firstName;
@property (nonatomic, retain) NSString * lastName;
@property (nonatomic, retain) NSDate * birthDate;
@property (nonatomic, retain) NSDate * dateOfEmployment;
@property (nonatomic, assign) Employee * manager;
@property (nonatomic, retain) NSString * ssn;
@property (nonatomic, readonly) NSTimeInterval age;
@property (nonatomic) double salary;

-(id)initWithFirstName:(NSString *)inFirstName
              lastName:(NSString *)inLastName
             birthDate:(NSDate *)inBirthDate ssn:(NSString *)inSsn;
-(id)init;
-(void)giveRaise:(double)percentage;
-(double)bonus;

@end
```

需要注意的第一点是，在这里指定的所有属性几乎都有它要映射到的数据成员。属性声明由@property 指令以及紧跟其后的特性组成，这些特性会影响作为属性的一部分创建的存取器函数类型。这些特性可以在@property 指令后的小括号内指定。对于给定属性可以指定的不同特性参见表 3-1。

表 3-1　属性特性

特　　性	功　　能
getter=<name>, setter=<name>	指定该属性所使用的存取器函数的名称
readwrite 或者 readonly	指定该属性是否可写。默认是可读可写
assign、retain 或者 copy	决定为该属性生成的赋值函数的类型。assign 生成的赋值函数是简单地为变量赋值。retain 生成的赋值函数在赋值到变量时会保留传入的参数。copy 生成的存取器函数会复制传入值到成员变量。默认值是 assign
nonatomic	指定生成的存取器函数是非原子性的，即非线程安全的。默认是原子性的，即线程安全的

紧接着属性特性必须指定属性的数据类型。属性不一定要直接映射到数据成员变量，如果是直接映射，数据成员的数据类型必须和此处指定的属性的数据类型匹配。最后，你必须指定属性名。你可以为属性指定一个和它所表示的实际数据成员不一样的名称。通常不需要这么做。因此，该名称需要和该属性映射到的数据成员名匹配。

通常，接口文件中声明的内容也要在实现文件中声明。属性也不例外。为了使用编译器自动生成的存取器方法，在实现文件的实现块中属性必须要有一个声明。实现文件中的属性声明的类型可以是@synthesize 声明或者@dynamic 声明。@synthesize 指令会使得编译器生成为属性创建存取器函数所需的所有代码。本质上，该指令是属性的“替代品”。如果使用@synthesize 指令，不需要在实现文件中为属性写任何代码。

另一方面，如果想手动创建存取器函数，现在或者之后动态加载到运行时环境中，可以通过使用@dynamic 指令来创建。在使用@dynamic 指令时，编译器会指望你为属性创建一对合适的存取器函数。



警告

使用@dynamic 指令创建存取器函数时，应该确保存取器函数履行了在属性的特性中指定的约定。换句话说，如果指定的是 copy 特性，就必须确保存取器函数在设置属性时复制传入的值。

代码清单 3-13 显示了员工类的实现。注意，这里有使用不同方式处理的不同属性。

代码清单 3-13 Employee 类实现

```
#import "Employee.h"


@implementation Employee
@synthesize firstName;
@synthesize lastName;
@synthesize birthDate;
@synthesize dateOfEmployment;
@synthesize manager;
@synthesize ssn;
@synthesize salary;
@dynamic age;

//删减了部分代码，这样就可以更关注属性

-(NSTimeInterval)age;
{
    return [birthDate timeIntervalSinceNow];
}


@end
```

大多数属性都利用@synthesize 指令定义。这就意味着编译器完全负责为这些属性创建存取器方法。在员工类中有几个属性完全是计算属性。比如员工的年龄可以通过员工的生日计算得到。从这段代码可以看到，在这种特定的情况下，这些属性已经被指定为“只读”和“动态”了。这意味着我们选择创建动态计算这些特性的方法而不是将它们存储成成员变量。可以从代码中看到，为此我们实现了执行计算的方法。

2. 合成的属性存取器函数

指定一个属性并通过灵巧的@synthesize 指令支持编译器生成自动合成的属性时，属性的特性会影响到存取器函数的行为。编译器本身实际上会根据这些特性生成不同的代码。

3. 使用 **nonatomic**

在声明属性时可以指定的一个重要特性就是属性存取器函数的原子性。属性的原子性和其在多线程环境下的行为有关。原子性的存取器函数可以确保该值完全是在访问它的线程中进行设置或者读取的。因此，原子性的存取器函数是线程安全的。本质上，由原子性的存取器函数生成的代码大致如代码清单 3-14 所示。

代码清单 3-14　一个原子性的存取器函数

```
-(NSString *)firstName
{
    [threadLock lock];
    NSString *result = [[firstName retain] autorelease];
    [threadLock unlock];
    return result;
}
```

非原子性的存取器函数不能被认为是线程安全的。一个通过`@synthesize` 指令生成的非原子性存取器函数如代码清单 3-15 所示。

代码清单 3-15　一个非原子性的存取器函数

```
-(NSString *)firstName
{
    return [[firstName retain] autorelease];
}
```

在确认只有一个线程访问对象的应用中可以选择使用非原子性存取器函数。

由于不需要原子性存取器函数中所需要的线程锁，使用非原子性存取器函数可以略微提高性能。

4. 使用 **assign**、**retain** 和 **copy** 特性

在属性特性中有一系列重要的特性用于指定生成的赋值函数的语义。它们分别是 `assign`、`retain` 和 `copy` 特性。这 3 个特性是互斥的，定义了与该属性一起使用的赋值函数的行为。

默认值 `assign` 指定将值简单地赋给数据成员。这种类型的例子如代码清单 3-16 所示。

代码清单 3-16　一个简单的 `assign` 风格的赋值函数

```
-(void)setFirstName:(NSString *)inValue
{
    firstName = inValue;
}
```

该特性通常用于标量属性、委托（delegate）以及不适合保留的其他类型的变量。

`retain` 属性特性只用于处理本身就是对象的数据成员。它指定在将传入到赋值函数的值赋给成员变量的同时向其发送一条保留消息。



说明

本节所使用的部分语言和下一章将会介绍的 Objective-C 内存管理相关。在读完那章后你可能还需要回过头来看看本节内容。

该风格的赋值函数的示例如代码清单 3-17 所示。

代码清单 3-17 `retain` 风格的赋值函数

```
-(void)setFirstName:(NSString *)inValue
{
    [firstName autorelease];
    firstName = [inValue retain];
}
```

最后，`copy` 特性指定所生成的赋值函数应该复制对象到成员变量。和 `retain` 类似，这仅仅在成员变量是对象时使用。通过 `copy` 特性风格的属性自动生成的存取器函数如代码清单 3-18 所示。

代码清单 3-18 一个 copy 风格的赋值函数

```
-(void)setFirstName:(NSString *)inValue
{
    [firstName autorelease];
    firstName = [inValue copy];
}
```

5. 属性名和数据成员名不一致

通常，属性名称和成员变量的名称一致。不过也有需要处理遗留代码的情况，这种情况下名字就可能不一致。在这种情况下，可以指定属性使用不同的存取器函数名称。代码清单 3-19 给出了一个例子。

代码清单 3-19 指定属性的存取器函数名

```
@property (nonatomic, retain, getter=getFirstName) NSString *firstName;
```



说明

Objective-C 存取器函数中的取值函数和赋值函数通常分别是 `variableName` 和 `setVariableName`。这是 Objective-C 的标准做法以使得对象符合键值编码规则。第 6 章将深入介绍这一主题。

3.2.2 使用点标记

内部实现上属性会编译成方法调用，在设置值时是赋值函数，在获取值时是取值函数。在使用 Objective-C 属性时，你可以直接使用传统的方法调用来使用这些赋值函数和取值函数，比如

[object setFoo:bar]，也可以使用一种称作点标记的特殊语法。代码清单 3-20 给出了一个同时使用传统的存取器函数以及点标记的示例。

代码清单 3-20　通过传统的存取器函数和点标记访问属性

```
{
    //传统的方法调用
    [employee setFirstName:@"John"];

    //新的 Objective-C 点标记
    employee.firstName = @"John";
}
```



警告

点标记仅在有定义了属性的值上可用。

C++、Python 和 Ruby 等一些语言使用点标记而不是属性来进行方法调用。如果你熟悉这些语言，你可能就会习惯使用点标记访问行为而不是状态，这是相当糟糕的做法。

3.3　应用对象

到目前为止我们已经详细了解了面向对象编程，现在我们可以一起创建一个展示面向对象编程技术的简单应用。

我们将要创建的应用是一个管理人力资源的应用。这是一个非常简单的应用。最终结果并不是一个可以实际使用的过程式应用，但是创建它的过程将会展示到目前为止介绍过的所有面向对象编程技巧。

首先创建一个命令行 Foundation 项目。

3.3.1　创建员工对象

该应用的作用就是存储员工和经理列表。此外，员工类支持给员工奖金、给员工加薪以及计算员工的年龄。

此外，有一种特殊类型的员工：经理。经理员工会有一些向他汇报的员工。员工会有他们经理的引用，这样他们就可以访问到经理。通过模板创建了该应用以后，创建一个名为 Employee 的新类。按照代码清单 3-21 编辑该类的接口。

代码清单 3-21　员工类接口

```
//
//  Employee.h
//  HR
//
```

```
//  Created by Jiva DeVoe on 4/22/10.
//  Copyright 2010 __MyCompanyName__. All rights reserved.
//

#import <Cocoa/Cocoa.h>


@interface Employee : NSObject
{
    NSString *firstName;
    NSString *lastName;
    NSDate *birthDate;
    NSDate *dateOfEmployment;
    Employee *manager;
    NSString *ssn;

    double salary;

}
@property (nonatomic, retain) NSString * firstName;
@property (nonatomic, retain) NSString * lastName;
@property (nonatomic, retain) NSDate * birthDate;
@property (nonatomic, retain) NSDate * dateOfEmployment;
@property (nonatomic, assign) Employee * manager;
@property (nonatomic, retain) NSString * ssn;
@property (nonatomic, readonly) NSTimeInterval age;
@property (nonatomic) double salary;

-(id)initWithFirstName:(NSString *)inFirstName
              lastName:(NSString *)inLastName
             birthDate:(NSDate *)inBirthDate
                   ssn:(NSString *)inSsn;
-(void)giveRaise:(double)percentage;
-(double)bonus;

@end
```

需要指出的是该文件的接口为员工类需要跟踪的属性定义了不同类型的数据成员。此外，我们还定义了访问这些数据成员的属性。其中的一些属性，比如年龄，不需要直接和数据成员相对应，它是可以计算的值。

大多数属性使用了 `retain` 特性，除了标量值、计算属性（只读）以及 `manager` 属性。`manager` 属性在本例中是个特殊的例子。`manager` 属性有一个向他汇报的员工列表。由于将员工加入到汇报列表中就会保留员工，所以就不应该将员工的经理属性也保留。原因将会在下一章介绍。目前，只需知道员工上的经理属性只能通过 `assign` 设置，而不能用 `retain` 设置即可。

创建了接口文件后，继续编辑实现文件，代码如代码清单 3-22 所示。

代码清单 3-22　员工类的实现文件

```
//
//  Employee.m
//  HR
```

```
//
//  Created by Jiva DeVoe on 4/22/10.
//  Copyright 2010 __MyCompanyName__. All rights reserved.
//

#import "Employee.h"


@implementation Employee
@synthesize firstName;
@synthesize lastName;
@synthesize birthDate;
@synthesize dateOfEmployment;
@synthesize manager;
@synthesize ssn;
@synthesize salary;
@dynamic age;

-(void)dealloc;①
{
    [self setFirstName:nil];
    [self setLastName:nil];
    [self setBirthDate:nil];
    [self setDateOfEmployment:nil];
    [self setSsn:nil];
    [self setManager:nil];

    [super dealloc];
}

-(id)init;
{
    if(self = [super init])
    {

    }
    return self;

}

-(id)initWithFirstName:(NSString *)inFirstName
              lastName:(NSString *)inLastName
             birthDate:(NSDate *)inBirthDate
                   ssn:(NSString *)inSsn;
{
    if(self = [self init])
    {
        [self setFirstName:inFirstName];
        [self setLastName:inLastName];
        [self setBirthDate:inBirthDate];
        [self setSsn:inSsn];
```

① 原文末尾的分号是多余的。——译者注

```
    }
    return self;

}

-(NSTimeInterval)age;
{
    return [birthDate timeIntervalSinceNow];
}

-(void)giveRaise:(double)percentage;
{
    salary = salary + (salary * percentage);
}

-(double)bonus;
{
    return salary * .05;
}

@end
```

需要指出的重要一点是,有一个接收多种参数的初始化函数用于初始化 Employee 类的不同成员变量。通过使用该初始化函数，创建的对象就可以初始化所有基本特性，以备使用。

另外一个要点就是在代码中我们实际创建了一个用于计算员工年龄的动态属性。为此，我们对 age 属性使用了@dynamic 指令。我们创建一个 age 存取器方法用于使用员工的生日计算年龄。这里还定义了两个方法，一个用于给员工加薪，另一个是给员工奖金。两个方法都会在经理类中重写以支持不同百分比的加薪和奖金。

3.3.2 创建经理类

这样我们就创建了 Employee 类，还需要创建一个 Employee 类的子类 Manager。该类包含一个向经理汇报的员工列表，并且它有不一样的加薪和奖金百分比。

为了创建经理类，需要在项目中新建一个类，编辑接口文件如代码清单 3-23 所示。

代码清单 3-23 经理类接口

```
#import <Cocoa/Cocoa.h>
#import "Employee.h"

@interface Manager : Employee
{
    NSMutableArray *reports;
}
@property (nonatomic, retain) NSMutableArray * reports;

-(void)addReport:(Employee *)inEmployee;

@end
```

回顾一下要创建继承自另一个类的类，需要在接口操作中的类名后指定父类名。可以从代码中看出就是这样做的。记住 Manager 类会继承 Employee 类所有的特性和行为。

现在你需要指定 Manager 这个类特有的项，确切地说，你需要添加一个报告数组来存储需要向该经理汇报的员工。此外，还需要一个添加报告人到报告列表的方法。切换到 Manager 的实现文件，按照代码清单 3-24 编辑该文件。

代码清单 3-24　manager 类的实现文件

```
#import "Manager.h"


@implementation Manager
@synthesize reports;

-(void)dealloc;
{
    for(Employee *employee in reports)
    {
        [employee setManager:nil];
    }

    [self setReports:nil];
    [super dealloc];
}

-(id)init;
{
    if(self = [super init])
    {
        [self setReports:[NSMutableArray array]];
    }
    return self;
}

-(void)addReport:(Employee *)inEmployee;
{
    [reports addObject:inEmployee];
    [inEmployee setManager:self];
}

-(double)bonus;
{
    return salary * .10;
}

@end
```

需要注意的是这里有一个用于创建并初始化报告列表的指定初始化函数。在调用 Employee 类的初始化函数时会调用指定初始化函数。如果需要在实现文件中访问 Employee 类，可以看到有一个[super init]方法调用。注意，bonus 方法在该类中也被重写了。奖励一个经理的奖金比率和奖励普通员工的不同。

最后需要指出的是，在 dealloc 方法中，当经理对象被释放时，它会遍历所有的报告者并将其经理属性设置成 nil。回顾一下，员工的经理特性使用 assign 特性而不是 retain 进行设置。所以，如果经理对象被释放了但向其汇报的员工没有，员工对经理的引用指针会变得无效。因此，经理类在释放自身时将汇报者的 manager 属性设成 nil 很重要。这是一个使用委托需要记住的重要模式。

3.3.3　在HR主函数中关联不同的类

现在我们已经完成了员工类和经理类的定义，接下来将要创建这些类的实例并将它们关联起来。

按照代码清单 3-25 编辑示例应用的主函数，然后就可以展示一些在现有类上可以进行的基本操作。

代码清单 3-25　关联员工类和经理类的实例

```
#import <Foundation/Foundation.h>
#import "Employee.h"
#import "Manager.h"

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    Employee *joeBlow = [[Employee alloc]
                                initWithFirstName:@"Joe"
                                         lastName:@"Blow"
                                        birthDate:
             [NSDate dateWithNaturalLanguageString:@"12/01/1990"]
                                              ssn:@"555-12-1212"];①

    Employee *janeDoe = [[Employee alloc]
                                initWithFirstName:@"Jane"
                                         lastName:@"Doe"
                                         birthDate:
              [NSDate dateWithNaturalLanguageString:@"11/01/1985"]
                                               ssn:@"555-12-1212"];

    Manager *johnAppleseed = [[Manager alloc]
                                initWithFirstName:@"John"
                                         lastName:@"Appleseed"
                                        birthDate:
             [NSDate dateWithNaturalLanguageString:@"11/01/1970"]
                                              ssn:@"555-12-1212"];
    [johnAppleseed addReport:joeBlow];
    [johnAppleseed addReport:janeDoe];

    joeBlow.salary = 50000;
    janeDoe.salary = 75000;
```

① 本例中所有的社保号都一样，有点不合理。——译者注

```
    johnAppleseed.salary = 100000;

    NSMutableArray *allEmployees = [NSMutableArray array];
    [allEmployees addObject:joeBlow];
    [allEmployees addObject:janeDoe];
    [allEmployees addObject:johnAppleseed];

    for(Employee *employee in allEmployees)
    {
        [employee giveRaise:.10];
        NSLog(@"Employee %@ %@'s salary is: %.2f with a bonus of: %.2f",
              employee.firstName, employee.lastName, employee.salary,
              employee.bonus);
    }

    [johnAppleseed release];
    [janeDoe release];
    [joeBlow release];

    [pool drain];
    return 0;
}
```

这段代码创建了 3 个员工，Joe Blow、Jane Doe 和 John Appleseed。John Appleseed 是 Joe Blow 和 Jane Doe 的经理。每个员工都在创建后加入到包含所有员工的数组中。最后该应用遍历员工数组，给他们奖金并加薪。

3.4 小结

本章介绍了面向对象编程的基础知识以及如何在 Objective-C 声明和使用类。我们学习了如何创建类、如何处理继承、如何处理多态、如何通过属性在类中封装数据。本书的剩余部分会更多地使用对象。

第 4 章

Objective-C 内存管理

本章概要

- 内存管理简介
- 使用引用计数
- 创建管理内存的对象
- 使用垃圾回收
- 转换现有代码以支持垃圾回收
- 理解需要使用什么样的内存管理模型

从 Java、Ruby 和 Python 等语言转向 Objective-C 平台的程序员新手要面临的最大的挑战之一，就是 Objective-C 需要考虑内存管理。很多其他现代语言都有内置的内存管理系统（比如垃圾回收）来帮助程序员完成大部分的内存管理问题。Objective-C 有一个垃圾回收的运行时版本，但对于该语言这还是一个较新的特性，而且在 iPhone 和 iPad 等平台上也不可用，所以对 Objective-C 新手程序员来说，尽管无需关注内存管理听起来会比较好听，但这对 Objective-C 的学习没有帮助。即使不在 Mac OS X 之外的平台上写 Objective-C 代码，你也很有可能会遇到不带垃圾回收、需要手动管理内存的 Mac OS X 代码。

幸运的是，如果熟悉了 Objective-C 的内存管理规则，即使手动管理内存也是很简单的。在完成本章的学习后，你应该会掌握操作内存管理代码以及非内存管理代码所需的所有工具的知识。

4.1　使用引用计数

在介绍 Objective-C 中用来管理内存的工具之前，首先介绍一下 Objective-C 在底层使用的一种机制，通过这种机制可使手动内存管理和使用垃圾回收环境几乎一样简单。

每个从 `NSObject` 继承的对象都继承了一定的内存管理行为。在这些对象的内部存在一个称作保留计数的计数器。在进行某些调用时，计数器的值可以增加或者减少。Objective-C 语言运行时知道当保留计数为 0 时，目标对象就可以被释放。在对象释放时，其所有的内存资源都会归还给系统以供重复使用。

保留计数可以通过几种标准的方式增加。最常见的方式是使用名字中包含 alloc 或者 create 的方法创建一个新对象，返回对象的保留计数是 1。此外，使用名字中包含 copy 一词的方法获取对象时，返回对象的保留计数也是 1。还可以通过调用 retain 方法手动增加保留计数。最后，可以通过调用 release 方法减少保留计数。再次强调，在保留计数变成 0 时，对象及其相应的内存都会被释放。

在下面的代码清单中，我用几个代码段示范实际应用。首先，在代码清单 4-1 中使用标准的对象分配和初始化方法创建一个对象。

代码清单 4-1　对象分配的标准方式

```
Bar *foo = [[Bar alloc] init];
```

在本例中，执行完该方法之后，foo 对象的保留计数是 1。

现在看看代码清单 4-2。

代码清单 4-2　保留对象

```
Bar *foo = [[Bar alloc] init];
[foo retain];
```

在本例中，除了通过调用 alloc 分配一个对象外，还通过 retain 方法增加了保留计数。执行完本段代码后，对象的保留计数是 2。

现在看看代码清单 4-3

代码清单 4-3　分配和释放对象

```
Bar *foo = [[Bar alloc] init];
[foo release];
[foo doSomething];
```

在本例中，调用 release 方法后，对象的保留计数变成了 0，从而会被释放。其后的代码由于试图访问被释放的对象而导致程序崩溃。

很显然，这样的代码是错误的。代码清单 4-4 给出了另一种错误的例子。这种情况下会发生内存泄漏。

代码清单 4-4　内存泄漏

```
Bar *foo = [[Bar alloc] init];
[foo retain];
[foo release];
```

在本段代码中，首先分配了一个对象，然后保留了该对象，之后释放它，但只释放了一次。如果这是一个成员变量，我会再次向它发送 release 消息，从而再次释放对象，这样就没有问题了。但在本例中，我们只在这个特定的栈中使用该代码，就会导致内存泄漏。

代码清单 4-5 显示了发生错误的另一个例子，你知道发生什么错误吗？

代码清单 4-5 没有保留对象

```
@interface Foo : NSObject
{
    memberVariable
}

@end

@implementation Foo

-(void)someMethod;
{
    memberVariable = [someOtherObject getFoo];
}

@end
```

在本例中，所讨论的代码应该保留通过方法名中不带 `alloc`、`copy` 或者 `create` 的调用获得的对象。在程序退出了该方法后，目标对象就会被程序运行时释放掉。下次访问该对象时就会导致崩溃。由于此处变量是作为成员变量使用的，所以应该保留。



说明

从技术上说，对象如果之前没有被保留，那么在下一个运行循环后会被释放。不过从遵循内存管理规则的角度考虑，你可以假定对象出了作用域后，只要没有被保留，就会被释放。

4.1.1 内存管理规则

记录保留计数看起来很复杂，但记住这些会使 Objective-C 的使用变得容易得多。

- 对于通过调用带有 `alloc`、`copy` 或者 `create` 一词的方法创建的任何对象及其内存，你都拥有所有权。你负责在之后的某个时刻向该对象发送 `release` 消息来释放资源。使用类似`[[Foo alloc] init...]`命令创建的对象需要释放。任何使用类似`[foo copy]`方法创建的对象需要释放。任何和 `CreateFoo()`类似的调用所返回的对象也需要释放。
- 对于通过不带有上述词的方法调用获得的对象，你都没有所有权。这些对象可以在当前执行栈中任意使用，离开当前栈以后，这些对象就不可用了。

通过其他方法调用获得的对象通常是“自动释放”对象。本章稍后会介绍自动释放，其关键在于，自动释放对象会在应用程序下次离开运行循环时被释放。这种释放很可能在离开当前方法后就发生。不要指望这些对象在当前方法之外还有效。比如，如果你要将其赋值给一个成员变量，就需要保留它。

无论是通过分配、复制还是保留来增加对象的保留计数，都会获得对象的所有权，并标记对象为你所有。这就是声明需要无限期地访问对象，在使用完成后就需要放弃对象的所有权，以使

其可以被销毁。



警告

尽管可以通过保留对象，成为一个对象的所有者，但所有权并不是独占的。其他人也可以拥有该对象。你不是唯一一个可以访问该对象并改变其值的人。

4.1.2　使用自动释放

之前已经提到过自动释放。自动释放的概念是 Objective-C 内存管理的核心。它使得 Objective-C 可以处理 C++和 C 等其他语言所面临的棘手问题：为从其他方法获得的对象定义一个标准的“转移”机制以及如何管理与其相关的内存。

比如试想一下，如果返回一个对象的 C++方法或者函数，那么谁是对象的所有者呢？是被调用的方法吗？还是方法的调用方呢？如何处理从一个所有者到另一个所有者的内存转移，而不需要内存完全公开在两者之间呢？C++和 C 通过不同的方式处理这类问题。大部分情况下，都是由各个程序员创建或记录要遵循的某种标准。结果就是在学习一个新库时，你还需要学习它所使用的内存管理系统。比如一些库倾向于使用智能指针，而其他库则倾向于使用一些已知的约定。

当 Objective-C 面临同样的问题时，Objective-C 的开发者提出了一个“自动释放”的概念。`autorelease` 是一个和 `release` 类似的可以在对象上调用的方法。然而，它并不能立刻减少对象的保留计数，你可以将 `autorelease` 看成是运行时环境的一种承诺，它会在下次应用的运行循环退出时减少保留计数。通常，这会在当前方法退出时进行。如果保留计数通过这种方式递减，对象会照常释放。

任何时候，返回通过方法名中不包含 `alloc`、`copy` 或者 `create` 一词的方法创建的对象，返回的对象都应该是自动释放的。`autorelease` 方法实际返回一个自动释放对象。因此，利用类似代码清单 4-6 所示的模式既方便又标准。

代码清单 4-6　返回一个自动释放对象

```
-(Foo *)getFoo
{
    Foo *foo = [[Foo alloc] init];
     //对 foo 执行一些操作
    return [foo autorelease];
}
```

另一种常见的有效使用自动释放的模式就是自动释放所创建的对象而不是手动释放。这样就无需担心所创建对象的内存管理，而让“自动释放池” 在方法退出后自动清理所遗留的任何对象。如代码清单 4-7 所示。

代码清单 4-7 alloc/autorelease 模式

```
-(void)someMethod
{
    Foo *foo = [[[Foo alloc] init] autorelease];

    //在这里 foo 仍然有效
    //在方法退出前不会被释放
    [foo doSomething];
}
```


你可能发现一个简单的释放模式：将对象的创建和释放放得非常靠近，使用这种模式时，忘记释放的可能性就很小了。此外，利用这种方式写代码会让你觉得 Objective-C 和 Python 或者 Ruby 等内存管理语言有几分相似。你可以认为这个特定栈帧中使用的所有变量在离开作用域后都会被释放。需要考虑的仅仅是在这个特定栈帧之外还需要保留的特定对象或者变量。

Cocoa 和 Cocoa Touch 框架提供了帮助你遵循该模式的其他工具。具体来说，`NSString`、`NSArray` 和 `NSDictionary` 等很多基础类都包括一个返回能自动释放的对象的工厂方法。使用这些方法而非 `alloc/init` 模式的构造函数，你几乎可以不用考虑内存管理了。

代码清单 4-8 展示了利用这类工厂方法的示例及其和传统模型的比较。

代码清单 4-8 使用工厂方法和使用传统创建模式的对比

```
-(void)usingFactories;
{
    NSMutableArray *array = [NSMutableArray array]; //漂亮简单的自动释放

    NSMutableArray *array2 = [[NSMutableArray alloc] init];
    //对 array 和 array2 进行相应操作

    //需要释放该对象
    [array2 release];

    //[array release];不要释放该对象
    //已经自动释放
    //如果在这里释放，将会导致崩溃
}
```

由于你把所创建对象的删除交由运行时处理，你就放弃了对何时删除对象的部分控制权。在理想的情况下，这些对象在下次运行时退出运行循环时就会被删除。但实际上，显然不会总是那么简单。由此需要避免创建大量使用自动释放池的对象。在一些有严格内存限制的平台，即使没有任何内存泄漏的情况下也可能耗尽内存，比如代码清单 4-9 所示的代码。

代码清单 4-9 在自动释放池中留有大量对象

```
-(void)inflateMemoryUsage
{
    for(NSUInteger n = 0; n < 100000; ++n)
    {
```

```
            //对象是自动释放的
               NSData *data = [self getBigBlobOfData];
            //对数据进行处理
            [self doStuff:data];
        }
        //在这里100 000个数据对象都还有效
    }
```

在本例中，代码看起来很简单。但是需要注意的是，我执行的是一个非常紧凑的循环，在该循环中分配了一些对象并将它们留在自动释放池中等待稍后释放。由于这是循环代码，执行程序流不会退出当前栈。自动释放池也就不会被清空。所以内存使用会一直攀升。

你可以通过几种不同的办法解决该问题。当然第一种可能就是直接释放对象，而不使用自动释放对象。比如，将代码清单 4-9 中的代码重写成代码清单 4-10 中的代码就不会有问题了。

代码清单 4-10　在循环中释放对象

```
-(void)inflateMemoryUsage
{
    for(NSUInteger n = 0; n < 100000; ++n)
    {

          //保留计数是1
        NSData *data = [[NSData alloc] init];
        [self putBlobOfDataIntoData:data];
        //使用创建的对象
        //处理数据
        [self doStuff:data];
        [data release]; //对象在这里被释放
    }
    //没有遗留任何东西
}
```

但是，有时你并没有办法控制是否用这样或那样的方式来释放循环中的所有对象。有时，比如由于库的原因，你会和代码清单 4-9 一样，在循环中使用很多自动释放对象。在这些情况下，合理的解决方法就是在循环内部创建一个自动释放池，在使用完后代码中的对象后，清空自动释放池并释放它。

如何实现正是下一节的内容。

自动释放池

创建一个新项目时，在自动生成的模板代码中你可能已经见过自动释放池了。如果你错过了，代码清单 4-11 给出了一个典型的 Foundation 命令行应用的示例。

代码清单 4-11　一个典型的 main 函数

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
```

```
        //在这里插入代码
        NSLog(@"Hello, World!");
        [pool drain];
        return 0;
    }
```

可以看到，该应用所做的第一件事就是创建一个 NSAutoreleasePool 对象来捕获所创建的并收到 autorelease 消息的所有对象。在主函数的末尾，自动释放池被释放之前会被清空。清空自动释放池实际会导致向所有自动释放的对象发送 release 消息。

所有应用都至少有一个 NSAutoreleasePool。如果应用有多个线程，每个线程都必须有各自的自动释放池。通常，大多数图形应用都有一个在每次运行循环执行前清空的自动释放池。这使得在应用运行时自动释放对象可以不断被释放，而不像代码清单 4-11 所示的仅在应用退出前释放对象。

要创建一个自定义自动释放池，你需要创建一个新的 NSAutoreleasePool 对象，并在随后执行需要的操作，包括自动释放任何需要自动释放的对象。当准备好实际释放已经自动释放的对象时，可以通过 drain 方法将释放池清空或者通过 release 释放它。该示例如代码清单 4-12 所示。

代码清单 4-12　创建一个自定义自动释放池

```
-(void)inflateMemoryUsage
{
    for(NSUInteger n = 0; n < 100000; ++n)
    {
        NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
        //该对象是自动释放的
        NSData *data = [self getBigBlobOfData];
        //对 data 进行些处理
        [self doStuff:data];
        [pool release];  //自动释放对象在这里释放
    }
    //没有遗留任何对象
}
```

自动释放池中自动释放对象被推入到可用的最高层级的自动释放池，这点和嵌套栈有些类似。所以如果在一个自动释放池内创建多个池，在清空池子时只有最里面的自动释放池内的自动释放对象被释放。此外，自动释放池是一种使得手动内存管理变得更简单的实用工具。不要将其同自动内存管理混淆。

下一节会介绍对象内部的内存管理以及如何确保对象正确地管理资源。

4.1.3　对象内部的内存

你可能想知道在分配和释放对象时对象内部会发生什么。当有人实例化你的对象时，他们会调用 init 方法。当它们释放对象时，dealloc 方法会自动被运行时调用。在创建并初始化一个新对象时，初始化函数需要分配并初始化该对象的所有成员变量。同样，释放一个对象时，dealloc 方法需要释放对象初始化函数分配的内存，以及执行对象的任何方法时所分配的所有动态内存。

根本上说，初始化一个新对象时，你会分配该对象所需的任何资源，而且在释放该对象时应该释放这些资源。



警告

永远不要自己调用 dealloc。

你已经看到了在创建一个新对象，将其赋给变量时，你调用了 alloc 类方法，并返回一个没有数据成员的分配对象。然后使用那个对象调用 init 或者其他适合你的类的初始化方法。初始化函数用来为对象中的成员变量分配内存。

你永远不要直接调用 dealloc 方法。它是你在对象上调用 release 方法时被间接调用的。dealloc 方法在对象被自动释放池释放时也会被自动调用。

如何在这些方法中分配和释放内存很重要。下节将详细介绍。

1. 编写初始化函数

编写初始化函数时，需要记得调用该类指定的初始化函数或者父类的指定初始化函数。

在调用当前类的指定初始化函数时，需要使用特殊变量 self，比如[self init]。要调用父类的指定初始化函数，你可以使用特殊的变量 super 并通过 [super init]来调用父类的指定初始化函数。这两个方法都会返回一个初始化对象 self，它代表你初始化的对象，并且必须从当前的初始化函数返回。如果在父类或者指定初始化函数的初始化中发生错误，就会返回 nil。

在编写正确的初始化函数过程中，重要但特殊的一步是，将父类或者指定初始化函数返回的 self 赋给自定义初始化函数中的 self 变量。这看起来有点像是错误，不过却是 Objective-C 的一个重要方面，你不应该回避。父类的初始化函数实际上可能会创建一个新的对象并作为 self 返回，而不是复用在当前类的初始化函数中创建的 self 对象。通常在存在类簇的情况下会这样做。类簇就是通过其中的一个子类实现的给定类。初始化函数确定要实例化的正确子类并将其返回。

在调用给定初始化函数并将结果赋给 self 后，你需要确认 self 不为 nil。如果在初始化过程中发生错误，就会从父类初始化函数中得到 nil。这种情况下就不要尝试初始化成员变量，而是也返回 nil。

只需一行代码就可以完成赋值和确认，如代码清单 4-13 所示。

代码清单 4-13　一个典型的初始化函数

```
-(id)init
{
    if(self = [super init])
    {
        someMemberVariable = [[Foo alloc] init];
    }
    return self;

}
```

在父类初始化成功后就可以分配并初始化成员变量了。这包括为所需要的成员变量调用所需的标准 Objective-C 初始化函数。在某些情况下，你可能想推迟创建某些成员变量，直至真正需要时再创建。在这种情况下就需要对代码进行相应调整。

在创建并初始化成员变量和其他资源后，就可以从初始化函数中返回 `self`。这样就满足了初始化函数的约定，将初始化的对象返回给调用者。

某些情况下，Cocoa 或 Cocoa Touch 框架在特定情况下会调用一些特殊的初始化函数。比如，在从 nib 文件反序列化一个对象时，就会调用特殊的初始化函数 `initWithCoder:`，从文件中解码序列化的类信息。这种情况很少见，但很重要。Objective-C 社区的成员目前正在争论在初始化函数和析构函数中使用 Objective-C 2.0 属性存取器函数来初始化成员变量是否合适，因为使用存取器函数会触发键值观查事件（后面的章节会介绍键值观查及其工作原理）。目前，我建议直接初始化成员变量，而不是在初始化函数和析构体中使用存取器函数。也就是说，即使无视这种建议也不会遇到问题。我经常在初始化函数和析构函数中使用存取器函数，但从没有遇到过问题。不过，通过这种方式可能会引入一个很隐蔽的 bug。

在使用 64 位运行时时，这种问题变得更复杂了，你可以在没有关联成员变量的情况下声明属性。在这种情况下，只能通过存取器函数来初始化或者释放成员变量。因此，苹果建议在使用 64 位运行时并且没有关联成员变量时使用属性时，应该在对象的初始化函数和析构函数中使用存取器函数，而在更早的 32 位运行时中不能在构造函数和析构函数中使用存取器函数。

2. 编写 `dealloc` 方法

为了释放在初始化函数中分配的内存，也必须编写一个析构函数。Objective-C 析构函数的方法名就是 `dealloc`。正如之前提到的，`dealloc` 是在调用 `release` 方法时被间接调用的。它在自动释放池清空，自动释放对象时也会被自动调用。`dealloc` 方法的示例如代码清单 4-14 所示。

代码清单 4-14　一个典型的 `dealloc` 方法

```
-(void)dealloc
{
    [someMemberVariable release];
    someMemberVariable = nil;
    [super dealloc];
}
```

在 `dealloc` 方法中应该释放对象所分配的任何资源，包括与其所有成员变量关联的内存。这可以通过在任何成员变量上调用 `release` 来实现。在释放一个成员变量后，务必要将之前存有成员变量数据的指针赋值成 `nil`。

和 Java 或者 C++等其他语言不同的是，Objective-C 规定在 `nil` 对象上调用方法结果是不执行任何操作（专业术语就是“无操作”）。因此在释放成员变量后将指针设置为 `nil` 是一个好主意。在释放成员变量后，尽管内存可能被释放，但指针依然指向之前对象存在的内存位置。其他数据可能会被立刻写入到相同的内存位置。在这种情况下，再次访问变量会导致崩溃或者无法预期的行为。如果忘记将成员变量设置成 `nil`，访问该成员变量就是访问内存中的未知值。这就是所谓的“野指针”（dangling pointer）。

使用将对象作为被委托的对象时也需要特别小心。在将对象作为其他对象的委托分配时，根据 Cocoa 标准不能保留被赋值对象（即委托）。

原因就是如果对象将另一个对象作为委托进行分配，结果会是循环保留，一个对象保留子对象，而子对象反过来将该对象作为委托保留。由于它们互相保留对方的引用，释放任何一个对象结果都会是两个对象都无法释放。因此，带有委托对象的对象应该将被委托变量指定为赋值而不是保留。这样在为一个对象释放委托对象时，可以在 `dealloc` 方法中把子对象的委托属性设置成 `nil`，以阻止子对象试图调用已经释放的委托对象的 `release` 方法。

另一种和委托问题类似的常见的情形就是观察者模式、或者是 Objective-C 对观察者模式的实现 `NSNotificationCenter`。使用 `NSNotificationCenter` 来通知对象特定事件时，你必须记着在对象释放时从 `NSNotificationCenter` 移除该对象的观察者。没有做到这点就会导致程序崩溃。这适用于一个对象观察另一个对象的键值变化的情形。如果你之前将自身设置为任一对象的键值观察者，务必要记得移除。第 6 章会介绍这方面的内容。

在释放资源并移除该对象的观察或者委托角色后，最后需要确保做的就是要调用父类的 `dealloc` 方法，这使得父类有机会释放它的资源。如果你没有调用父类的 `dealloc` 方法，就会产生内存泄漏。`[super dealloc]`调用应该一直都是 `dealloc` 方法的最后一行，这样就可以在清理完自身内存分配后调用。

4.2　使用垃圾回收

在应用中手动管理内存的想法听起来很烦琐，你一定很高兴听到最近苹果在 Objective-C 上添加了垃圾回收机制的新闻。你可能不熟悉垃圾回收这一概念，垃圾回收是应用的运行时用来动态确定哪些对象在应用中不再使用或者引用，并自动释放那些对象的方法。使用了垃圾回收的应用不需要担心释放对象、保留周期，或者几乎不用担心内存泄漏。合理使用垃圾回收可以帮助避免一些程序员新手经常遇到的问题。

遗憾的是垃圾回收有个限制。垃圾回收只在 Mac OS X 10.5 或者更高版本上可用。目前在 iPhone、iPad 或者 Linux、Windows 等平台上不可用。因此，学会如何编写适合手动管理内存环境的 Objective-C 代码很重要。

此外，在内存管理方面尽管垃圾回收可以解决很多问题，但它对你高效使用垃圾回收提出了额外的挑战。

4.2.1　垃圾回收器

如果你是从 Java、Python 或者 Ruby 等其他语言转向 Objective-C，你很可能已经熟悉垃圾回收这一概念了。了解 Objective-C 垃圾回收器的工作原理可以帮助你编写适合 Objective-C 环境的代码。

了解 Objective-C 垃圾回收器函数的基本工作原理很重要。对于任何带有内置运行循环的应用，比如 Cocoa 或者 Cocoa Touch 应用，在主运行循环执行时垃圾回收器就开始运行，并查找没有活跃引用的对象。垃圾回收器找到一些就会释放它们。

原理就是垃圾回收器在应用中查看一组根对象并查找所有包含根对象的引用。所有不能从根对象访问到的对象都被视为“垃圾”并会被回收。根对象被定义为全局变量、栈变量和外部引用。

比如，主应用的全局实例引用的任何对象，或者被主应用的全局实例所引用的任何对象引用的任何对象都不能当做垃圾回收的引用。但是，如果一个对象在方法中赋值给该方法中的另一个变量并在方法退出后不再引用，那么最初保存它的变量已经出了作用域。该对象引用现在保存在内存中并且没有从根对象可以到达的引用。该对象可能就会被回收。

如果一个对象被回收，其内存也会被释放，但是不会调用 dealloc 方法。在垃圾回收环境中，dealloc 方法是过时的并且不会再使用到。为此，引入了一个名为 finalize 的新对象方法。finalize 方法和 dealloc 方法有很多相似之处，比如它是在对象被删除之前最后被调用的方法，大多数情况下它是进行对象所需的任何清理的正确位置。但是，由于垃圾回收的工作方式，finalize 方法有一些 dealloc 方法所没有的限制。是否为对象添加 finalize 方法是可选的。实际上，尽量不要使用 finalize 方法。

回顾一下 dealloc 方法的主要作用是手动释放一些为成员变量分配的内存。由于在垃圾回收环境中不需要做这些，因此也就没有必要在 finalize 方法中进行类似处理。因此，使用 finalize 方法的唯一原因就是释放其他有限的资源。（遗憾的是，基于我之后将介绍的原因，finalize 方法实际上是一个释放有限资源的糟糕的地方。）

由于垃圾回收器在何时调用 finalize 方法方面是相对不确定的，或者说无法确定它何时被调用。因此，如果需要释放的资源很重要，你可能需要将释放这些资源的代码放在另一个可以在确定时间调用的方法中，而不是依赖垃圾回收器以及它何时可能会释放你的对象。

垃圾对象被回收的顺序也是不定的。因此，你在 finalize 方法中调用其他对象都有失败的可能，这取决于你的对象或你调用的对象是否被先释放。

基于这些原因，应该避免使用 finalize 方法。如果没有其他选择，本章稍后会介绍如何编写 finalize 方法的合适的方法以及如何在该方法中管理有限资源。

回顾一下，任何应用都有一个运行循环，Cocoa 和 Cocoa Touch 应用会自动拥有一个垃圾回收器（假设你已经启用了项目中的垃圾回收功能）。但如果你正在编写一个基础应用，比如本书到目前为止所编写的应用，则你必须在应用的主函数中手动实例化并启动垃圾回收器。

在本书接下来的部分，大多数情况下我会使用 Cocoa 的 GUI 应用作为示例项目。但是，我还是想展示一下在 Foundation 应用中如何启动垃圾回收器。为此，你需要调用如代码清单 4-15 所示的 objc_startCollectorThread()函数。

代码清单 4-15　在基础应用中使用垃圾回收器

```
int main (int argc, const char * argv[])
{
    objc_startCollectorThread();

    // ...

    return 0;
}
```

引用类型

为了真正理解如何在应用中使用垃圾回收，你需要理解影响应用中垃圾回收工作方式的两个基本概念。这两个概念就是强引用和弱引用。

Objective-C 中的所有对象指针都是引用，意味着它们指向了为对象分配的内存。一个引用可以是强引用或者弱引用。默认情况下，所有的 Objective-C 引用都是强引用。强引用是就是垃圾回收器跟踪的引用，以确认一个对象是否存在，不应该被回收。而弱引用就是可赋值给对象的有效引用，但在所引用的对象没有强引用的情况下允许被垃圾回收。

如果你想引用一个被标记为要被释放的对象，并且不想保留它时，这个概念就很有用了。比如，NSNotificationCenter 使用了对所注册的观察者的弱引用。记住在引用计数的环境中，注册为 NSNotificationCenter 观察者的对象在各自的 dealloc 方法中必须移除观察者的身份。在垃圾回收环境中，这点不是必须的，因为当对象被释放时，NSNotificationCenter 中该对象的弱引用就变得无效并且被设置为 nil。因此，NSNotificationCenter 不再会试图向该对象发送通知了。

当你看到这里时会想着这应该是创建委托时所使用的合适的模式。记住，在一个引用计数的环境中，一个对象有委托时，就会使用 assign 属性特性来有效创建一个委托的“弱”引用。这样做的原因就是防止循环保留。在垃圾回收环境中，垃圾回收器可以检测到并防止循环保留，这时没有必要对委托使用弱引用。

正如之前提到的，Objective-C 中所有的引用默认都是强引用。为了定义一个弱引用就要使用 __weak 关键字。该示例如代码清单 4-16 所示。

代码清单 4-16　定义一个弱引用

```
@interface Foo : NSObject
{
    __weak NSString *memberVariable;
}

@end
```

除了手动指定一个给定的引用为弱引用外，还有一些特殊的容器类可以用于存储一系列弱引用。你可以在需要存储弱引用时将 NSArray 或 NSDictionary 等类替换成这些类。

这些类是 NSMapTable、NSHashTable 和 NSPointerArray。使用这些类时，如果数组中的一个元素被释放，该元素的引用也会从数组中移除。通常，使用 NSArray 等类时，数组的强引用可以保持对象“活着”。

4.2.2　为项目配置垃圾回收

为项目配置垃圾回收也相对比较简单。需要做的就是配置编译设置，并改变 Objective-C 垃圾回收的设置。如图 4-1 所示，如果要搜索编译设置中的该设置，你可以点开所有可能值的下拉列表。这些值如下所示。

- ❑ Unsupported：使用保留计数的内存管理系统。
- ❑ Supported：指定项目支持垃圾回收但不需要。
- ❑ Required：应用及其所有使用的框架都需要垃圾回收。

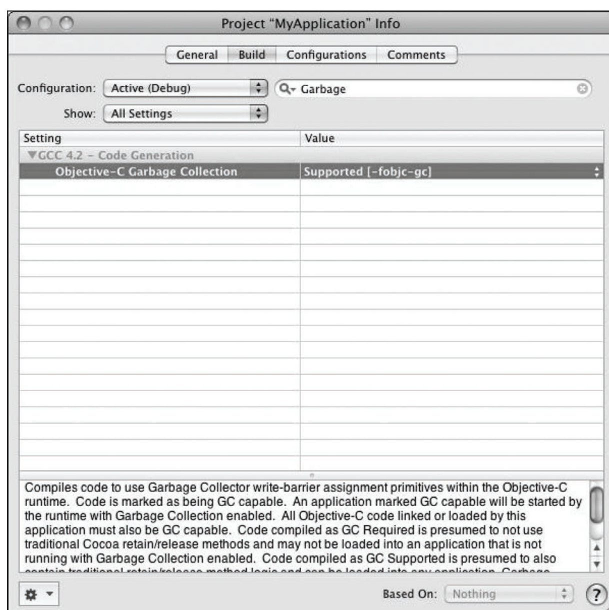


图 4-1　垃圾回收的编译设置

第一种设置就是没有垃圾回收。可以通过选择 Unsupported 选项启用该设置。第二个选项是 Supported。该选项添加了`-fobjc-gc` 标志来指定项目支持垃圾回收但不需要。使用该设置时，你可以将项目链接到没有带针对垃圾回收进行编译的应用。这个设置通常仅用于库。使用该设置，代码要求同时实现 `dealloc` 方法以及 `finalize` 方法，因此不论要链接的应用编译时是否支持垃圾回收都可以使用。

最后一种设置 Required 会在编译设置上添加一个`-fobjc-gc-only` 编译器标志，指定你的代码不使用 `retain/release` 方法，并且不能加载到不支持垃圾回收的应用中。

如果你将一个现有的应用从非垃圾回收环境转换到垃圾回收环境，你必须知道和非垃圾回收内存管理模型相关的所有方法都不再适用。初始化方法还是会被调用，但 `dealloc` 方法就不会。因为你必须重构 `dealloc` 方法，要么删除任何资源释放的代码，要么将其移到 `finalize` 方法。

4.2.3　在垃圾回收项目中使用框架

在应用中使用垃圾回收，还必须确保任何链接到的框架或库编译为支持垃圾回收。正如之前所展示的，在编译一个支持垃圾回收的框架或者库时，你必须选择只支持在一种环境中使用，支持在两种环境中使用（框架可以同时在支持垃圾回收和不支持垃圾回收的应用中使用），或者不

支持垃圾回收。对于后一种情况，库或框架就无法链接到应用。

所幸的是，所有 Cocoa 框架都完全支持垃圾回收。会遇到不支持垃圾回收的情况就是使用第三方库。在 Mac OS X 上的 Objective-C 支持垃圾回收已有一段时间了，因此不支持它的库应该很少。

4.3　关键的垃圾回收模式

在创建支持垃圾回收的应用时会不可避免地遇到一些设计模式。认识这些模式并了解处理问题的方式可能会很有帮助。

4.3.1　管理有限的资源

面向对象语言中一个经常使用的设计模式就是为有限的资源编写一个对象包装器。在处理文件、套接字等内容时，使用对象包装器是一种常用的做法。这个模式的优势就是资源可以在对象的初始化函数中创建并在该对象的 `dealloc` 方法中释放。这就提供了一个确保分配的对象可以被释放的概念模型。

在垃圾回收环境中该设计模式存在几个问题。首先当然就是 `dealloc` 方法不会被调用。因此如果除了启用垃圾回收之外没有进行任何变更，试图包装并确保被释放的有限资源将永远不会被释放。

这在对象是操作系统资源，比如套接字的情况下尤其会导致很大问题。将释放从 `dealloc` 方法移到 `finalize` 方法可能很诱人（我已经提到过目标应该是不使用任何 `finalize` 方法）。记住在垃圾回收环境中，对象可能在未来某个时刻释放。换句话说，你不能指望在应用执行的某个时刻会调用 `finalize` 方法。因此，希望在 `finalize` 方法中释放的资源可能在程序执行了很长一段时间之后才释放。

如果两个问题都考虑，你就需要重新考虑该设计模式并多做一些工作，包括对象本身以及使用该对象的对象以确保分配的资源在被垃圾回收之前被适当关闭。

展示这个概念的最简单方式就是看代码。代码清单 4-17 就显示了一个典型的文件包装器类，该类分配了一个文件句柄资源，也就是它的初始化函数并在 `dealloc` 方法中释放该资源。

代码清单 4-17　一个典型的文件包装器类

```
@interface Foo : NSObject
{
    int fileHandle;
}

@end

@implementation Foo

-(id)init
{
    if(self = [super init])
```

```
    {
        fileHandle = open(...);
    }
    return self;

}
//方法在这里

-(void)dealloc
{
    close(fileHandle);
    [super dealloc];
}

@end
```

将该类转变成一个合适的可以被垃圾回收的类，需要将文件句柄的释放从 `dealloc` 方法中提出来，并放在一个可以被对象的使用者手动调用的方法中，比如 `close` 方法。

代码清单 4-18 显示了一个可以在垃圾回收环境中使用的更新后的类。

代码清单 4-18　可以被垃圾回收的文件包装器类

```
@interface Foo : NSObject
{
    int fileHandle;
}

@end

@implementation Foo

-(id)init
{
    if(self = [super init])
    {
        fileHandle = open(...);
    }
    return self;
}

-(void)close;
{
    if(fileHandle != -1)
        close(fileHandle);
    fileHandle = -1;
}

-(void)finalize;
{
    [self close];
    [super finalize];
}
@end
```

4.3.2　编写支持垃圾回收的基础应用

之前简单介绍了一下编写支持垃圾回收的 Foundation 命令行应用这一主题。你应该已经知道了要创建一个使用垃圾回收的 Foundation 命令行应用，必须在 `main` 函数开始的地方，分配任何对象之前启用垃圾回收器。然而，还有一个需要了解的关键细节。

Objective-C 垃圾回收器的工作方式就是它会在全局栈和当前本地栈上查找指向对象的指针。在这个过程中，它会查看本地栈中当前所有的活动变量。在查看过程，它不会考虑本地栈变量是否被初始化。记住，一个变量在初始化之前指向的内存地址以前可能包含对象或变量的初始化数据，而这些数据已经被删除。换句话说，当前栈中未初始化的变量可能指向一个在之前的函数调用中分配的对象，并且这个对象没有再被引用。因此，垃圾回收器可能会误认本地变量引用了一个本该被垃圾回收的对象。

为了防止这种问题出现，需要定期清理本地栈。需要使用的底层 Objective-C 运行时方法是 `objc_clear_stack(OBJC_CLEAR_RESIDENT_STACK)`。通常，在命令行 Foundation 应用中，框架提供的应用并不包含运行循环，你必须提供自定义的运行循环。运行循环的顶部被认为是清理本地栈并防止这类问题出现的理想地方。

考虑到这两件事，一个典型的命令行 Foundation 应用的 `main` 函数应该和代码清单 4-19 类似。

代码清单 4-19　命令行 Foundation 应用 `main` 函数

```
int main (int argc, const char * argv[])
{
    objc_startCollectorThread();

    // ...

    while(running)
    {
        objc_clear_stack(OBJC_CLEAR_RESIDENT_STACK);

        for(RunnableItem *item in runnableItems)
        {
            [item run]; // ..
        }
    }

    return 0;
}
```

4.3.3　处理nib文件中的对象

在应用中使用 nib 文件中的对象，并且该对象不包含指向其中的实例化对象的引用时，有时候会遇到常见很难调试的问题。需要再次指出的是，垃圾回收器会查找没有引用全局对象或者当前栈中的对象的任何对象。在一些极少见的情况下，你可能创建一个 nib 文件，该文件包含一些没有被引用这些对象的对象，比如，没有外部引用的视图控制器。通常情况下，这些对象是会被

垃圾回收的。该问题的解决方案就是创建一个拥有该 nib 文件的 `IBOutlet`，并将这个插座连接到所涉及的对象。这会为对象提供一个强引用并防止其被垃圾回收。



说明

nib 文件或者“NeXT Interface Builder”文件是用来在 iOS 和 Mac OS 上定义界面的。本书不会讨论这些，因为这仅仅与 Cocoa 和 Cocoa Touch 相关。要了解更多信息，请参照 Wiley 的 *Cocoa Developer Reference or Cocoa Touch for iPhone OS 3 Developer Reference* 一书。

4.3.4 强制垃圾回收

在某些情况下，你可能会在应用执行过程中的某一时刻强制垃圾回收器回收任何可以被回收的对象。比如，如果你刚分配然后释放了大量的对象，比较合适的做法就是告诉垃圾回收器开始回收，这样那些对象就可以尽快被完全释放。

为此可以使用底层方法 `objc_collect()`。该方法强制垃圾回收器开始一个回收循环。我会在本章详细介绍该方法。



说明

当需要将不支持垃圾回收的代码转换成支持垃圾回收的代码或者在两种环境中都可以工作的代码，自动释放池的存在是在暗示回收器，它也是可以回收的。这是垃圾回收器的一个未记录但众所周知的功能。

4.3.5 处理空指针和垃圾回收

在支持垃圾回收的应用中，另一个常见的 Objective-C 模式可能造成的困难就是，使用 `void*` 数据类型在回调函数间传递应用特有的数据。该方法的优势是被传递的数据可能开发人员想要的任意类型的数据，可能是一个对象，或者可能是字节块。接收方法类型将 `void*`参数转换成任何它希望接收的数据类型。

在垃圾回收环境中会产生问题的原因是由于空指针是不透明的，所以可能被误认为是一个没有引用的对象，因此会被垃圾回收。这可能在调用方法和回调函数之间发生，从而导致回调方法接收到的指针最终会指向一个已经释放的对象。

该问题的解决方案就是使用 Core Foundation 方法 `CFRetain` 和 `CFRelease` 来强制实现一个在 Core Foundation 框架中全局维护的强引用。代码清单 4-20 给出了一个具体的例子。

代码清单 4-20 保留不透明的指针，以用于回调

```
@implementation Baz

-(void)callbackMethodForObject:(id)object withUserInfo:(void *)inData
{
```

```
    Bar *bar = (Bar *)inData;

    //对 bar 执行一些操作

    CFRelease(bar);
}


-(void)startLongOperation
{
    Bar *bar = [[Bar alloc] init];
    CFRetain(bar);
    foo = [[Foo alloc] init];
    [foo startLongOperationWithDelegate:self callbackMethod:
     @selector(callbackMethodForObject:withUserInfo:)
                               userInfo:bar];
}

@end
```

本质上，在将用户数据指针传递给将调用你的回调的对象之前，可以手动调用 CFRetain，然后作为参数传入用户数据指针。这样就在目标对象上建立了一个强引用。

接着，在调用回调函数并传递用户数据指针后，你必须再次调用 CFRelease，同时传入指针作为它的参数。这会移除对指针的强引用，这样一来垃圾回收器下次执行时，如果你没有创建该对象的其他强引用则回收该对象。

4.3.6　使用垃圾回收的面向对象接口

除了适合在底层代码中使用的垃圾回收器的函数接口外，苹果还提供了一个可以用于垃圾回收器的高级抽象，即 NSGarbageCollector 类。

NSGarbageCollector 类是一个单例类，支持通过一个 Objective-C 接口和垃圾回收器交互。可以调用 defaultCollector 方法访问当前线程的垃圾回收器。得到垃圾回收器的单例实例后，就可以通过它启用或者禁用某些指定指针的回收，甚至整个线程的回收。此外，你还可以使用它通过 collectExhaustively 或者 collectIfNeeded 方法强制回收。

NSGarbageCollector 类的方法如表 4-1 所示。

表 4-1　经常使用的 NSGarbageCollector 方法

方　　法	功　　能
+defaultCollector	返回当前线程的 NSGarbageCollector 单例
-disable/-enable	暂时禁用或者启用垃圾回收
-isEnabled	如果当前启用了垃圾回收则返回 YES，否则返回 NO
-collectExhaustively	触发一次彻底的回收
-collectIfNeeded	只有在由于上一次回收而导致内存消耗超过阈值时才触发垃圾回收
-disableCollectionForPointer:	将一个指针作为根对象，使其不会被回收
-enableCollectionForPointer:	将给定指针从根对象列表移除，使其可以被回收

4.4 项目使用的内存管理模型

理解垃圾回收的关键在于什么时候应该使用和什么时候不应该使用。在很多情况下在应用中使用垃圾回收不是一个好选择。正如之前提到的，这只在 Mac OS X 10.5 及其之后的版本上可用。如果你的代码在 iPhone 或 iPad 等其他平台上运行，根本没有垃圾回收。并且，如果某个应用并没有使用垃圾回收，其中大量使用了依赖于引用计数式内存管理的代码，那么就没有什么必要将它转换为支持垃圾回收的代码了，因为代价太高昂了。

根据应用中某一时刻有效对象的数目，垃圾回收器可能效率会相对较低，释放的对象直到回收器找到它们后才真正被释放。此外，垃圾回收器本身必须利用 CPU 周期来进行工作。如果这是你要考虑的，在应用中使用垃圾回收可能就不是一个合适的选择。

但是，在应用中使用垃圾回收也有很大的优势。比如通常来说，使用垃圾回收的应用更容易实现线程安全。这是因为存取器函数变成不再需要线程锁实现线程安全的简单赋值操作。

支持垃圾回收的应用通常也比较容易编写。不需要维护一个对委托的弱引用，这样就会有更容易维护的更简单的代码。

最后，可以忽略引用计数的内存管理所需的常见样板代码是一个很大的进步。有人说过最好的代码和 bug 最少的代码就是根本没必要写的代码。当然，垃圾回收通过减少所需编写的代码量来减少 bug 出现的机会。

记住，你必须使用可以达成目标并满足性能要求的最高层面的抽象。也就是说如果可以使用垃圾回收，并且问题域性能也可以的话，应该使用。如果垃圾回收在特定问题域性能欠佳就不要使用。Objective-C 为开发者提供的一个最大优势之一就是可以从框架栈向上或向下，使用最容易解决问题的合适等级的抽象。

4.5 小结

本章可能是本书中最重要的章节之一。理解内存管理技术以及引用计数和垃圾回收的合理应用是 Objective-C 程序员的一项关键技能。在本章中，我首先介绍了 Objective-C 的传统内存管理模型，引用参数。介绍了如何在对象中分配内存，如何在不再使用时释放内存。还介绍了如何使用 Mac OS X 可用的新技术，垃圾回收。与传统的引用计数型内存管理相比，垃圾回收可以使得代码更简单并且 bug 更少。你必须自己确定哪种内存管理技术比较适合你的项目。不过，我希望我已经给你足够的工具来有效地做出那个决定。

Part 2 第二部分

更多特性

本部分内容

第 5 章　代　码　块

本章概要

- ❑ 使用代码块封装算法
- ❑ 使用代码块指令
- ❑ 通过代码块创建映射和过滤函数
- ❑ 使用线程或者统一中央调度并发运行代码块

Objective-C 的一个最新也是最强大的改进就是加入了一种称作代码块（block）的功能。通过它们，你就能够像对待对象一般，指定要在方法和函数中传递的任意代码部分。本章将展示如何使用它们。

5.1　了解代码块

如果你是从 Ruby 或者 Lisp 等其他语言转向 Objective-C 的，那么可能已经熟悉代码块（也称作闭包）这一概念了。代码清单 5-1 显示了一个 Ruby 中代码块的示例。

代码清单 5-1　Ruby 中的代码块示例

```
items.each { |item| puts item }
```

本质上，代码块支持在代码中内联地定义一个函数对象。这些函数对象可以通过传统的变量来引用，包括将它传入到其他函数。这就意味着，你可以定义可复用的代码段，并且可以像对象一样到处传递从而动态地在其他对象内部执行。这听起来可能有点令人迷惑，但通过接下来的几个例子，你就能对这个概念有更清晰的认识了。

在上述代码块中，代码实际上是先遍历数组中的各个元素，然后执行大括号内部的代码，并将当前元素传入到代码块中。

5.1.1　声明代码块

在本节中，我会介绍 Objective-C 中的代码块。代码清单 5-2 显示了一个简单的代码块的示例。

代码清单 5-2　Objective-C 中的简单代码块示例

```
int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    void (^myBlock)(NSString *x);

    myBlock = ^(NSString *x)
    {
        NSLog(@"%@", x);
    };

    [pool drain];
    return 0;
}
```


代码块本质上是一个和其他变量类似的变量。所不同的是，代码块中存储的数据是一个函数体。使用代码块时，你可以像调用其他标准函数那样调用代码块函数，传入参数并得到返回值。

对于本段代码，保存代码块的变量名为 `myBlock`。首先，通过 `void (^myBlock)(NSString*)` 声明变量。普通变量的声明相对简单。普通变量不需要传入参数，并且没有返回值。代码块则是存储在一个变量中，并且需要参数和声明的返回类型。因此，代码块的声明比传统变量的声明要复杂。

代码块声明包括返回类型（本例中是 `void`）。声明代码块的返回值类型的位置与所声明变量的类型定义在同一个地方。在代码块声明中声明的值类型就是代码块执行时的返回值类型。

紧接着返回值类型定义的是一个特殊操作符，它告诉编译器所定义的是代码块而不是其他类型的变量。这个操作符就是`^`字符。

你会发现将它看做指针变量的声明可能会更容易一些。正如声明指针变量时使用`*`字符标识目标变量是指针一样，代码块只不过是使用`^`字符而已。

在`^`字符之后，给出了存储代码块的变量名（`myBlock`）。这个变量名使用小括号同其后的参数隔开。

代码块变量的命名规则同其他变量的命名规则一样。它必须仅包含字母数字，但不能以数字打头。

在代码块变量名的右括号之后，用另外一对小括号列出了需要传入到代码块的参数列表，参数之间以逗号分隔（本例中是`(NSString *x)`）。在列出参数时，无需提供参数的变量名。是否提供变量名由你来决定，但这不是必须的。也许这样理解较好：目前还没有声明函数体，所以提供参数的变量名没有任何作用，因为暂时不会用到它。你仅需告诉编译器参数的类型即可，多个参数类型要以逗号隔开。很多代码块的文档在声明时都省略了参数名，但我不会这样做，因为我觉得这样的代码会使人迷惑，特别是对于新手而言。

和往常一样，使用分号结束语句。到目前为止，我们声明了一个变量，用于存储代码块，它接收指定参数并返回指定的值类型。代码清单 5-2 中的变量名为 `myBlock`。

众所周知，仅仅声明变量是不够的。存储代码块的目的是为了使用它。通过赋值操作符就可以利用代码块初始化一个新的变量，接下来，这个特别的语法表示创建的是需要存储到变量的实

际代码块。

代码块的定义再次使用^字符，来告诉编译器接下来的内容是代码块定义。在定义中可以省略返回值类型，因为编译器可以从存储代码块的变量确定返回值类型。但是必须再次在括号中提供代码块的参数说明。在本例中，你必须提供传入参数的变量名。这也是理所当然的，因为这里为参数声明的变量名会在代码块的主体中使用。

在参数列表的右括号后面，需要提供代码块的函数体。代码的函数体和声明一个普通函数几乎一致。函数体使用大括号括起来，可以执行指定的任何操作，在需要时使用参数并在结束后返回适当的值。和定义标准函数一样，代码块中的代码可以跨多行，普通的空格规则也适用。

代码清单 5-3 显示了一些不同类型的代码块定义。

代码清单 5-3　不同类型的代码块定义

```
void (^myBlock)(NSString *x) = ^(NSString *x)
{
    NSLog(@"%@", x);
};

void (^anotherBlock)(NSString *x) = ^(NSString *x) { NSLog(@"%@", x); };

void (^aVoidBlock)() = ^{ NSLog(@"blah"); };

doIt(^(NSString *x){ NSLog(@"%@", x); });
```



说明

如果所声明的代码块不需要任何参数，语法上允许不提供用于隔开参数的小括号，因为此时并不存在参数。不过在定义的时候还是需要提供小括号的。

如代码清单 5-3 所示，在一个表达式内可以同时进行代码块变量的声明和初始化。这点也和使用普通的变量一样。你可以先声明变量，之后再初始化，也可以一次完成。

在代码的最后一行，你可以看到程序允许内联地定义一个代码块来代替任意需要它的参数，同样可以传入一个硬编码的值作为参数代替变量参数。这是完全合法的。

5.1.2　使用代码块

声明代码块的主要原因就是可以在任何地方使用它。因此了解如何将代码块传入到其他函数或者方法，以及在接收到代码块时如何使用代码块对象至关重要。

要声明接受代码块参数的函数或者方法，需要在代码中按照声明代码块变量的方式声明一个参数。比如，代码清单 5-4 展示了一个接收代码块作为参数的函数。该代码块接收一个 `NSString` 参数并返回 `NSComparisonResult`。

代码清单 5-4　声明一个接收代码块参数的函数

```
void useCodeBlock(NSComparisonResult (^theBlock)(NSString *value));
```

在使用代码块的函数体内，可以通过代码块的变量调用代码块，变量就和普通的函数名一样。换句话说，你可以将该变量当作函数使用，括号内是代码块所需的输入参数，使用赋值操作符存储返回值。

代码清单 5-5 显示了一些之前定义的函数，并演示了如何使用所传入的代码块。

代码清单 5-5 使用了代码块的函数

```
void useCodeBlock(NSComparisonResult (^theBlock)(NSString *value))
{
    if(NSOrderedSame == theBlock(@"foo"))
        doSomethingIfSame();
    else
        doSomethingElse();
}
```

将代码块作为参数传入到对象或类方法（不是函数的参数）时，语法稍有不同。代码清单 5-6 演示了它的实现方式。

代码清单 5-6 将代码块传入到对象方法

```
-(NSMutableArray *)filterArray:(NSArray *)inArray
                     withBlock:(BOOL (^)(NSInteger))block
{
    NSMutableArray *result = [NSMutableArray array];
    for(NSNumber *number in inArray)
    {
        if(block([number integerValue]))
            [result addObject:number];
    }
    return result;
}
```

注意，我们是在代码块定义之后才传入了代码块参数的名称（保存代码块的变量在方法体内使用的名称），因此，通常在代码块定义时需要提供代码块变量名的位置却仅传入了^符号。

Objective-C 的这个新特性很强大。你可以创建比之前更灵活、可复用性更高的代码。

但是，在使用这个强大的语言特性时，需要注意几个细节。

5.2 了解重要的代码块作用域

如果只使用传入的参数并且只返回所定义的值，那它本身就已经是一个很强大的功能了。但是，代码块还有使其更强大的底层工具。

当在另一部分代码内定义一个代码块时，所定义的代码块，即代码块内的指令不仅可以访问到其他代码都能够访问的普通全局变量，而且可以自动得到所有栈变量的只读副本，这些栈变量的作用域为代码块定义时所在的栈。这就意味着在程序运行时，代码块拥有对所定义位置的程序的完整状态的只读访问权限。

这方面的示例参见代码清单 5-7。

代码清单 5-7　从所定义的栈上访问变量的代码块

```
int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSString *formatStr = @"%s";

    void (^myBlock)(char *x) = ^(char *x){ NSLog(formatStr, x); };

    doIt(myBlock);

    [pool drain];
    return 0;
}
```

可以看到，代码块访问了一个没有传递给它的变量（`formatStr`），但该变量可以从代码块创建时所在环境中得到。

需要指出的是，这类变量在代码块中是只读的。但是在引用代码块中，声明变量时可以通过特殊的语言指令`__block`显式地将变量声明为可读写的。

由于代码块具备了给应用状态"拍快照"的能力，并通过这种方式使其在应用的其他地方可用，它就提供一个可以封装和操作数据的极其强大的机制。

我还想继续展示代码块的其他一些使用方式，不过在此之前需要解决一些小问题。

5.2.1　管理代码块内存

在 Objective-C 中代码块和其他东西一样，也是对象。实际上，构成代码块的数据和普通变量类似，都是在栈上分配的。因此，如果将一个代码块传递给另一个函数或者对象，那个对象为了以后使用该代码块就需要保存它，接收对象必须保留代码块，这和接收一个传入的对象类似。

代码清单 5-8 显示了代码块如何工作的示例。

代码清单 5-8　在成员变量中保存代码块的对象

```
@interface Foo : NSObject
{
    void (^myBlock)(NSString *);
}
-(void)doSomethingWithBlock;
-(void)setMyBlock:(void (^)(NSString *))inBlock;
@end;

@implementation Foo

-(void)dealloc;
{
    [myBlock release];
    [super dealloc];
```

```
}

-(void)setMyBlock:(void (^)(NSString *))inBlock
{
    myBlock = [inBlock copy];
}

-(void)doSomethingWithBlock
{
    myBlock(@"foo");
    // ....
}
@end
```


可以看出，和其他任何 Objective-C 对象类似，所有标准的 Objective-C 引用计数内存管理方法在代码块上都适用。由于它是在栈上分配的，因此对于传入的代码块对象需要使用`-copy`而不是`-retain`，如果需要保留它就必须在堆上得到一个副本。

工作机制是，运行时会将代码块使用的任何外部变量以及`self`对象都以常量的方式复制到堆上，这样就你可以访问那些变量以及所有成员变量（代码块中创建的对象的成员变量）了。任何通过`_block`指令标识的变量都会按位复制到堆上，代码块就需要负责与使用这些变量相关的其他内存管理工作。

如果应用使用了垃圾回收，而不是引用计数的内存管理，那么复制、保留和释放就都无需你操心了。

5.2.2 通过typedef提高代码块的可读性

如果使用`typedef`关键字进行代码块定义，有时会使代码更易读。这使得你不必重新敲入代码块的所有参数和返回类型就可以复用这些定义。代码清单 5-9 显示了上一节的类，不过这次为代码块参数使用了`typedef`。可以看出，代码变得更清晰，可读性也提高了。

代码清单 5-9 使用 typedef 的相同代码

```
typedef void (^BlockWithCharArg)(char *);

@interface Foo : NSObject
{
    BlockWithCharArg myBlock;
}
-(void)doSomethingWithBlock;
-(void)setMyBlock:(BlockWithCharArg)inBlock;
@end;

@implementation Foo

-(void)dealloc;
{
    [myBlock release];
    [super dealloc];
```

```
}

-(void)setMyBlock:(BlockWithCharArg)inBlock
{
    myBlock = [inBlock copy];
}

-(void)doSomethingWithBlock
{
    myBlock("foo");
    // ....
}
@end
```

5.3　在线程中使用代码块

我相信你可以想到代码块的不同用途。在接下来的几节中我们还会介绍几个。

试想一下代码块支持将应用中的功能封装成一个漂亮干净的包从而轻松地复用该功能，这同样也适用于提供并行运行的代码的这一 Objective-C 中最常用的代码块的设计模式。换句话说就是多线程。

事实上，苹果在 Objective-C 中引入代码块时所展示的第一个用例就是能够在 GCD（Grand Central Dispatch）这一当时全新的并行框架中使用它们。

5.3.1　使用GCD

GCD 是一个随 Mac OS X 10.6 发布的框架。它提供了一个易用的抽象层，这样开发者不需要处理底层的线程管理就可以充分利用多处理器和多核架构。

通过 GCD，开发者只需要提供代码块，这些代码块封装了可以安全地并行执行的功能。使用时将代码块加入到 GCD 队列中，而后者会处理线程创建、线程管理，甚至应该创建多少个线程来运行在给定系统上提供的任务等底层细节。GCD 知道一个机器有多少个内核并分配足够的线程来最大化内核的性能。它可以完全处理交给它的排队的任务并将这些任务交给所创建的线程。

任务可以作为函数或者代码块传入到 GCD。显然，考虑到本章的主题，我们重点介绍在 GCD 中使用代码块。

5.3.2　使用GCD在线程中调度代码块

GCD API 的核心是队列的概念。通过 GCD，可以选择预先存在的系统队列，如全局队列。可以通过 `dispatch_get_global_queue()`方法访问到它，该方法返回一个和应用关联的全局并发队列，或者可以通过函数 `dispatch_queue_create` 创建一个私有的顺序执行的队列。

队列本身可以是并发的或顺序执行的，也就是对应于并发队列和顺序队列，队列中的对象相对于其他对象分别是并发执行和顺序执行的。



说明

在要求独占访问资源时以执行一系列操作时，经常使用顺序队列。通常，你需要使用一个线程锁来保证对这些资源的独占访问。通过 GCD，你就可以将不同的操作推入到一个顺序队列中。每个条目都只能在执行完后之前条目后执行。

调度代码块到队列中很简单。代码清单 5-10 显示了一个使用了执行耗时操作的代码块示例。在这种情况下，要在全局并发队列中调度该代码块。可以看出，我们可以将多个这类任务调度到该队列。GCD 可以自动处理任务的调度和管理，而不需要任何人为介入。

5

代码清单 5-10　调度代码块到全局并发队列

```
dispatch_async(dispatch_get_global_queue(0, 0), ^{ doSomethingSlow(); });
```

通过 `dispatch_get_global_queue` 方法就可以获得全局队列，然后将代码块 `(^{ doSomethingSlow(); } )`调度到该队列中。这里，在需要时可以使用之前介绍的任何代码块模式。当然，通常的线程安全要求也适用。

5.4　通用的代码块设计模式

之前我们介绍了如何在 GCD 中使用代码块将工作单元调度到线程。现在看看其他常见的代码块模式。在这些情况中，有一些标准框架的 API 使用代码块比不使用代码块能更高效更简洁地处理问题的示例。

5.4.1　将代码块作为映射

在其他语言中展示代码块威力的一个常见操作就是实现映射算法。

如果你没有印象的话，那么我再介绍一下。映射其实就是一个函数，它对数组的每个元素应用一个给定的函数并返回一个结果列表。利用代码块在 Objective-C 中实现映射极其简单。首先，需要创建一个映射函数。该映射函数接收代码块以及元素的数组作为参数，如代码清单 5-11 所示。

代码清单 5-11　使用代码块的映射函数

```
NSArray *map(NSArray *items, id (^block)(id item))
{
    NSMutableArray *result = [NSMutableArray array];

    for(id item in items)
    {
        [result addObject:block(item)];
    }

    return result;
}
```

使用映射函数时只需要构建代码块对象和数组，并将它们传入到映射函数中，如代码清单 5-12 所示。

代码清单 5-12　调用映射函数

```
NSArray *mappedResults = map(items, ^(id item){ return transformItem(item); });
```

5.4.2　在标准API中使用代码块

创建自定义的映射函数确实很强大,但与一些接收代码块参数的标准的 Cocoa 框架 API 结合使用时，代码块的真正威力才发挥出来。

一些 Cocoa 中的类接收代码块参数来执行操作，通常应用于其所管理的集合中的元素。常见的类包括 NSArray、NSDictionary、NSIndexSet 和 NSSet。此外，NSString 和 NSAttributedString 也提供了使用代码块逐行或逐个属性遍历的方法。

表 5-1 列出了一系列 Cocoa 中利用代码块的最常用方法。

表 5-1　Cocoa 中利用代码块的常用方法

类	方　法	功　能
NSNotificationCenter	addObserverForName:object:queue: usingBlock:	在收到通知时执行给定的代码块
NSIndexSet	enumerateIndexesInRange:options: usingBlock:, enumerateIndexesUsingBlock:, enumerateIndexes WithOptions:usingBlock:	遍历集合的索引，执行给定代码块，同时将索引传入到代码块
NSDictionary	enumerateKeysAndObjectsUsingBlock:, enumerate KeysAndObjectsWithOptions:usingBlock:	遍历字典的键和对象，执行给定的代码块，同时将键和对象作为参数传入代码块
NSString	enumerateLinesUsingBlock:, enumerateSubstrings InRange:options:usingBlock:	遍历字符串的行，调用给定的代码块同时传入当前行作为参数
NSArray	enumerateObjectsAtIndexes:options: usingBlock:,enumerateObjectsUsingBlock:, enumerateObjects WithOptions:usingBlock:	遍历数组的元素，同时将每个元素传入到给定代码块
NSSet	enumerateObjectsUsingBlock:, enumerateObjects WithOptions:usingBlock:	遍历集合的元素，同时将每个元素传入到给定代码块
NSOperationQueue	addOperationWithBlock:	将给定代码块加入到队列
NSBlockOperation	+blockOperationWithBlock:	利用给定的代码块创建一个新的 NSOperation 对象

最有意思的应该是 NSOperationQueue 和 NSBlockOperation 的方法，它们支持将 NSOperation 和代码块一起使用。相对于 5.3.2 节介绍的 GCD 函数，这是一个更高级的 API。第 16 章将介绍如何使用这个 API。

5.5　在易并行任务中应用代码块

到目前为止我已经介绍了和如何在代码中使用代码块相关的所有知识，现在就可以应用这些知识了。尽管代码块的使用不仅限于并行化和线程处理，但是它们擅长处理这些任务。

在编程界，这些就是大家熟知的*易并行问题*——唯独并行化才能很好处理的问题。

该类问题的一个示例就是素数计算。尽管还有其他一些比暴力计算更快的素数计算方法，但出于如下示例的考虑，我会展示如何改进暴力计算方法的性能。因此，我会和大家一起编写一个简单的程序，用于计算 2 ~ 150 000 的所有素数。然后通过两个不同的方式重写该应用。第一种方式使用一个代码块和 `NSArray`，展示如何封装判断给定数是否是素数的代码。第二种方法利用 GCD 来并行化计算。示例代码可以输出所找到的素数。如果想看看工作过程可以随时取消对相关代码的注释。不过主要的还是要注意每个方法需要花费多少时间。程序会输出每个方法所需要的时间。你应该可以看到原始直接的方法应该和数组过滤方法花费的时间相当，而并行版本会快得多。

5.5.1　创建项目

要做的第一件事就是创建一个项目。（到目前为止你应该做过几次了，所以这里就不赘述了。）创建一个新的命令行 Foundation 项目。尽管不一定非要这样，我将要创建一个进行素数计算的类。使用 Objective-C 进行编程时大多数情况都会使用对象和类。因此建议从现在开始就熟悉它们并在日常工作中使用它们。

创建项目后，按代码清单 5-13 修改 main 源码文件。

代码清单 5-13　素数计算器的 main 源文件

```
#import <Foundation/Foundation.h>
#import "PrimeFinder.h"
int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    PrimeFinder *finder = [[PrimeFinder alloc] initWithMaxNumber:150000];
    [finder start];

//如果需要打印出所有素数可以取消注释
//    for(NSNumber *number in [finder primes])
//    {
//        NSLog(@"Found prime: %@", number);
//    }

    NSLog(@"Found all the primes in %fs", [finder elapsedTime]);

    [finder release];
    [pool drain];
    return 0;
}
```

示例应用的三个版本中该源码文件都一样。在修改完该源码文件后，你可以创建一个用于素数计算的类。在项目中选择新建一个类并将该类命名为 PrimeFinder。在第一版应用中不会使用代码块，这样就可以体验一下用旧办法时程序是什么样子的。

PrimeFinder 的接口和实现分别如代码清单 5-14 和代码清单 5-15 所示。

代码清单 5-14　PrimeFinder 的接口文件

```
#import <Cocoa/Cocoa.h>


@interface PrimeFinder : NSObject
{
    NSInteger maxNumber;
    NSDate *startedDate;
    NSDate *endedDate;
    NSMutableArray *primes;
}
@property (retain, nonatomic) NSMutableArray * primes;
@property (retain, nonatomic) NSDate * startedDate;
@property (retain, nonatomic) NSDate * endedDate;
@property (readonly) NSTimeInterval elapsedTime;
-(id)initWithMaxNumber:(NSInteger)inMaxNumber;
-(void)start;


@end
```

代码清单 5-15　PrimeFinder 的实现文件

```
#import "PrimeFinder.h"


@implementation PrimeFinder
@synthesize startedDate;
@synthesize endedDate;
@synthesize primes;
@dynamic elapsedTime;

-(void)dealloc;
{
    [primes release];
    [startedDate release];
    [endedDate release];
    [super dealloc];
}

-(id)initWithMaxNumber:(NSInteger)inMaxNumber
{
    if(self = [super init])
    {
        maxNumber = inMaxNumber;
        primes = [[NSMutableArray alloc] init];
```

```
    }
    return self;

}
-(BOOL)isPrime:(NSInteger)number
{
    for(NSInteger n = 2; n < number; ++n)
        if((number % n) == 0)
            return NO;
    return YES;
}


-(void)start
{
    [self setStartedDate:[NSDate date]];

    for(NSInteger n = 2; n <= maxNumber; ++n)
    {
        if([self isPrime:n])
            [primes addObject:[NSNumber numberWithInteger:n]];
    }

    [self setEndedDate:[NSDate date]];
}

-(NSTimeInterval)elapsedTime
{
    return [endedDate timeIntervalSinceDate:startedDate];
}

@end
```

在代码清单5-15中，代码很简单。主要是一个从2一直到150 000这个我们所传入的最大数之间的循环，每次都调用 `isPrime` 方法来检查这些数是否是素数。如果该方法返回真，就将该数加入到结果集合中。`isPrime:`函数接收任何传递给它的数字，然后试图用小于该字的数除它。如果可以整除，也就是没有余数，那么该数就不是素数。但如果穷尽了所有小于自身（1除外）的数也无法整除，那它就是一个素数并返回真。

编译并运行程序后，就可以在终端输出计算机计算这些素数所需的总时间。在我的计算机上，它花了15秒。如果你的计算机比我的快得多，就可能花更少的时间，你就可以将150 000换成一个更大的数来增加最大的素数备选数。这是很重要的，出于本例的目的，计算机至少要花一定的时间来处理该问题。但不要加到太大。计算素数是很耗 CPU 的，如果太大就会需要很长时间才能完成。

5.5.2　在数组中使用代码块过滤素数

这个首次展示代码块使用的例子实际上并没有带来任何性能改进。展示它只是为了说明这是可以用来解决其他类型问题的一个有用的设计模式。

本质上所要更改的是修改 PrimeFinder 类，使其可以接收备选素数序列中所有可能的素数并将其放入到数组中。你可以编写一个方法，利用传入到过滤方法的代码块来过滤数组。该过滤方法会返回一个包含素数的新数组。这种模式可用于从数组中过滤满足一定条件的元素的情况。本例中使用代码块的好处就是，在任何时候仅通过向过滤函数传入不同的代码块就可以动态地改变条件。



说明

实现此功能的另一种方法就是利用 NSPredicate 类方法+predicateWithBlock:创建一个 NSPredicate 实例，并在 NSArray 方法 filteredArrayUsingPredicate:中使用它。再次说明，这里是介绍基础概念，因此我们选择使复杂的方法来实现。

代码清单 5-16 显示了本版程序中接口文件所需的变化。主要变更就是添加了 candidates 数组。

代码清单 5-16　过滤版本的 PrimeFinder 的接口文件

```
#import <Cocoa/Cocoa.h>


@interface PrimeFinder : NSObject
{
    NSInteger maxNumber;
    NSDate *startedDate;
    NSDate *endedDate;
    NSMutableArray *primes;
    NSMutableArray *candidates;
}
@property (retain, nonatomic) NSMutableArray * candidates;
@property (retain, nonatomic) NSMutableArray * primes;
@property (retain, nonatomic) NSDate * startedDate;
@property (retain, nonatomic) NSDate * endedDate;
@property (readonly) NSTimeInterval elapsedTime;
-(id)initWithMaxNumber:(NSInteger)inMaxNumber;
-(void)start;


@end
```

主要变化点都在代码清单 5-17 所示的实现文件中。

代码清单 5-17　过滤版本的 PrimeFinder 的实现文件

```
#import "PrimeFinder.h"


@implementation PrimeFinder
@synthesize startedDate;
@synthesize endedDate;
@synthesize primes;
```

```
@synthesize candidates;
@dynamic elapsedTime;
-(void)dealloc;
{
    [self setCandidates:nil];
    [self setPrimes:nil];
    [self setStartedDate:nil];
    [self setEndedDate:nil];
    [super dealloc];
}

-(id)initWithMaxNumber:(NSInteger)inMaxNumber
{
    if(self = [super init])
    {
        maxNumber = inMaxNumber;
        candidates = [NSMutableArray new];
        for(NSInteger n = 2; n <= inMaxNumber; ++n)
        {
            [candidates addObject:[NSNumber numberWithInteger:n]];
        }
    }
    return self;

}

-(NSMutableArray *)filterArray:(NSArray *)inArray
                     withBlock:(BOOL (^)(id))block
{
    NSMutableArray *result = [NSMutableArray array];
    for(id item in inArray)
    {
        if(block(item))
            [result addObject:item];
    }
    return result;
}

-(void)start
{
    [self setStartedDate:[NSDate date]];

    BOOL (^isPrime)(id) = ^(id number)
    {
        NSInteger value = [number integerValue];
        for(NSInteger n = 2; n < value; n++)
            if((value % n) == 0)
                return NO;
        return YES;
    };

    [self setPrimes:[self filterArray:candidates withBlock:isPrime]];

    [self setEndedDate:[NSDate date]];
```

5

```
}
-(NSTimeInterval)elapsedTime
{
    return [endedDate timeIntervalSinceDate:startedDate];
}

@end
```

首先，在初始化函数中创建一个 candidates 数组。

创建 candidates 后，你需要改变 start 方法。在本版本中，所接收的 isPrime:方法不是 PrimeFinder 类的对象方法，而是作为代码块创建的。

你还必须创建针对 candidate 数组和代码块进行调用的 filter 方法，该方法返回过滤后的数组，该数组中的元素均为素数，所以代码块返回真。在代码清单 5-17 中，该方法是 -filterArray: withBlock:。

该段代码的神奇之处就是可以向过滤函数传入任何代码块。它会逐个遍历数组成员，然后对每个元素调用代码块，并将该元素作为参数传递给代码块。代码块可以干任何事情。它要做的就是在目标元素应该在结果数组中时返回真，否则返回假。将该逻辑和遍历数组分离的能力很强大并且在一些情况下很有用。

5.5.3　使用GCD

PrimeFinder 类的最终版本利用 GCD 来并发运行素数计算。稍后我会更加详细地介绍 GCD。目前，只要知道我们会将检查每个数是否是素数的代码块实例加入 GCD 全局队列进行调度。

为此，需要进行一些处理，以确保对不同线程间公用的任何变量的访问都是安全的。这会影响到一些方面。首先，你需要将所有的结果，即素数存储到一个名为 result 的数组中。由于该数组是所有代码块公用的，必须在 start 方法自身的作用域内声明它。回顾下当声明一个代码块时，它会接收创建时所在栈上的所有局部变量和状态。但是，这些变量都是只读的，包括 result 数组。不过不需要使用_block 指令，因为尽管 result 变量是只读的，但它包含的内容是指针引用，所以不是只读的。这是应该注意的微妙但重要的一点。

除了让 result 数组可写外，我们还确保没有两个代码块同时写入数组。为此我们使用了一个 Objective-C 内置的简单线程安全机制，即@synchronized 关键字。

最后，为了实际在 GCD 上调度代码块，需要创建一个调度组，这样你就可以将代码块放入完全由 GCD 管理的全局队列。它会自动根据运行机器的内核数生成合适数量的线程，然后逐一从队列移除代码块并放入到这些线程中执行。

代码清单 5-18 显示了为此需要对 PrimeFinder 的实现进行哪些修改。请按如下代码修改该类。



说明

如果你是基于前一个例子的项目，记得从接口中删掉 candidates 数组。

代码清单 5-18　GCD 版本的 PrimeFinder 的实现文件

```
#import "PrimeFinder.h"


@implementation PrimeFinder
@synthesize startedDate;
@synthesize endedDate;
@synthesize primes;
@dynamic elapsedTime;

-(void)dealloc;
{
    [self setPrimes:nil];
    [self setStartedDate:nil];
    [self setEndedDate:nil];
    [super dealloc];
}

-(id)initWithMaxNumber:(NSInteger)inMaxNumber
{
    if(self = [super init])
    {
        maxNumber = inMaxNumber;
    }
    return self;

}

-(void)start
{
    [self setStartedDate:[NSDate date]];

    NSMutableArray *result = [NSMutableArray array];


    dispatch_queue_t globalQueue = dispatch_get_global_queue(0, 0);
    dispatch_group_t group = dispatch_group_create();
    for(NSInteger number = 2; number <= maxNumber; ++number)
    {
        dispatch_block_t isPrime = ^
        {
            for(NSInteger n = 2; n < number; ++n)
                if((number % n) == 0)
                    return;

            @synchronized(result)
            {
                [result addObject:[NSNumber numberWithInteger:number]];
            }
        };

        dispatch_group_async(group, globalQueue, isPrime);
    }
```

```
    dispatch_group_wait(group, DISPATCH_TIME_FOREVER);

    [self setEndedDate:[NSDate date]];
    [self setPrimes:result];
}

-(NSTimeInterval)elapsedTime
{
    return [endedDate timeIntervalSinceDate:startedDate];
}

@end
```

在本段代码中，目标代码块的类型被指定成 `dispatch_block_t` 类型，这是 GCD 函数为定义传给 GCD 队列的代码块而提供的一个特殊的 `typedef`。理解的要点就是这和之前使用过的代码块类似。`dispatch_block_t` 的实际定义如代码清单 5-19 所示，你可以参考。

代码清单 5-19　dispatch_block_t 的定义

```
typedef void (^dispatch_block_t)(void);
```

如果运行该版本的程序，你就会发现性能的实质性改进。在我的计算机上，计算素数所需的运行时资源节省了 45% ~ 50%。当然在你的计算机上这些数据会有所不同。

5.6　小结

本章介绍了 Objective-C 工具箱中一个强大的新工具。代码块非常有用，它可以封装一小块匿名的代码并且像对象一样传递这些代码块。这使得仅将新类型的代码块作为参数传入，就可以创建一个经过改进具备不同功能的更通用的方法。此外，代码块使得 GCD 变得极其简单，因为它提供了表达功能块并将功能传入到队列进行执行的能力。

第 6 章

键值编码和键值观察

6

本章概要

- ❑ 学习键值编码
- ❑ 编写符合 KVC 的存取器方法
- ❑ 利用 KVC 简化复杂任务
- ❑ 通过键值观察来观察其他对象的变化
- ❑ 实现手动或者自动的 KVO 通知

Objective-C 运行时提供了很多高级工具，利用它们不仅可以同操作系统框架交互，还可以同代码的特性交互。本章要介绍的工具称作键值编码。键值编码（或者 KVC）经常会在 Objective-C 语言中提及。

6.1 通过键值编码访问对象属性

除了一般的赋值方法和取值方法之外，借助键值编码，你还可以用一些标准化存取器方法来访问类的特性。通过指定表示你要访问的属性名的字符串标识符，可以使用这些存取器方法获取或设置类的属性。除了使用字符串标识符访问类特性外，你还可以使用标准化语法获取对象关系和子对象。

代码清单 6-1 给出了一个例子。

代码清单 6-1　一些示例类

```
Some example classes.

@interface Bar : NSObject
{
    NSArray *array;
    NSString *stringOnBar;
}
@property (retain, nonatomic) NSArray * array;
@property (retain, nonatomic) NSString * stringOnBar;
@end
```

```
@interface Foo : NSObject
{
    Bar *bar;
    NSString *stringOnFoo;
}
@property (retain, nonatomic) Bar * bar;
@property (retain, nonatomic) NSString * stringOnFoo;
@end
```

上述代码的两个类中有一个名为 Foo 的类。该类有一个字符串特性和一个定义了 Foo 和 Bar 类之间关系的特性。这种关系通过 Foo 类的 bar 属性来定义。

之前我们介绍过如何给一个指定类的特性定义属性，Objective-C 为每个属性提供了赋值方法和取值方法。除了标准的赋值方法和取值方法外，还提供了一套键值编码存取器方法。

键值编码存取器方法中最经常使用的是用于直接访问指定类特性的方法。-valueForKey: 可以通过指定一个参数来读取特性，该参数用字符串表示你要访问的特性名。-setValueForKey: 用于设置一个给定特性的值，也需要指定字符串作为特性名。

在处理更复杂的关系时，比如，访问一个特性的特性，你就需要使用点标记指定一个更复杂的键路径。比如，如果有一个对象，它有一个 Bar 类型的特性 bar，而 bar 又有一个名为 stringOnBar 的特性，那么就可以使用方法 -valueForKeyPath:，指定点标记路径为 "bar.stringOnBar"特性。此外还有一个-setValue:forKeyPath:方法。

看演示代码可能最容易理解上述内容。通过刚才介绍的函数调用，Objective-C 运行时会自动生成代码来读取或改写刚才提到的这些类的特性，如代码清单 6-2 所示。

代码清单 6-2　使用 KVC 存取器方法访问类的特性

```
Foo *foo = [[Foo alloc] init];
[foo setValue:@"blah blah" forKey:@"stringOnFoo"];
NSString *string = [foo valueForKey:@"stringOnFoo"];
[foo setValue:@"The quick brown fox." forKeyPath:@"bar.stringOnBar"];
NSString *string2 = [foo valueForKeyPath:@"bar.stringOnBar"];
```

可以看出，从字面上说可以通过一个字符串指定想要访问的特性名，这样就可以获取特性的值或者改写它的值。如代码清单 6-2 所示，你甚至可以访问 Bar 类的特性，它遍历两个对象之间的关系并访问你所访问的主对象的子对象的特性。

这初看起来有点令人困惑，你可能会想为什么我需要了解这些呢。对于大部分的日常编码来说，与直接使用类的赋值方法和取值方法相比，使用键值编码来设置或访问属性可能要敲入更多的代码，并且也更容易出错。但是，在少数一些特殊情况下，需要动态访问某些特性时，使用一些可以在运行时而不是编译时改变的值就尤为强大了。

代码清单 6-3 显示了需要将 pen 对象序列化到数据库表的一个类。在所展示的第一个例子中，序列化函数需要遍历表的各个字段。在遍历时需要一个复杂的 if 语句来确定需要在对象上使用哪个存取器方法，这样在序列化时才能把值序列化到给定字段中。

代码清单 6-3 不使用 KVC 序列化表

```
-(BOOL)serializeToTable:(Table *)inTable
{
    Row *row = [inTable addRow];
    for(Column *column in row)
    {
        if([[column name] isEqualToString:@"firstName"])
            [column setValue:[self firstName]];
        else if([[column name] isEqualToString:@"lastName"])
            [column setValue:[self lastName]];
        else if([[column name] isEqualToString:@"age"])
            [column setValue:[self lastName]];
        else if([[column name] isEqualToString:@"birthDate"])
            [column setValue:[self lastName]];
        ...
    }
    [row save];
}
```


这样的代码很快就会变得一团糟。现在看一下使用键值编码方法来实现同样功能的代码吧！代码清单 6-4 显示了更新后的代码。

代码清单 6-4 使用 KVC 的序列化方法

```
-(BOOL)serializeToTable:(Table *)inTable
{
    Row *row = [inTable addRow];
    for(Column *column in row)
    {
        [column setValue:[self valueForKey:[column name]]];
    }
    [row save];
}
```

可以看出，新版本不仅减少了代码行数，而且可扩展性更强、更灵活。如果需要在表中加入新的列，只需为对象添加一个属性来存储该列的值即可。这个序列化方法可以自动从特性中提取那些值并进行存储，而无需对方法进行任何修改。我要再次强调的是，序列化方法不是能在代码中到处使用的方法。我要强调，KVC 存取器方法通常比普通的存取器方法需要敲入更多的代码。不过在某些情况下，利用它可以编写出动态性更强的代码，并减少需要表示对象特性信息的代码。当查询一个对象的特性时，你不必在代码中的多处地方重复提供这些信息。这是很好的——没有 bug 的代码就是还没有写出的代码。

6.1.1 键路径

为了使用键值编码，你必须先弄明白如何构建键路径以及用该键路径可以访问到什么。

你可以将 KVC 存取器方法想象成一个字典。字典的键都是字符串。字符串本身是要操作的对象特性名。由于这些要求，键以及特性命名就必须遵循一定的规则。首先，键必须是 ASCII

编码的。这就意味着键不能使用不常见的字符，这种字符在特性名中通常也不会使用。第二，键必须以小写字母打头。第一个字母可以是下划线，但不能是数字，也不能是大写字母。第三，键不能包含空格。

所幸，由于访问的属性必须是一个有效的符号名，因此不太可能违反这些规则。

代码清单 6-5 显示了一些有效的键路径和无效的键路径。

代码清单 6-5　有效和无效的键路径

```
//有效
[foo valueForKeyPath:@"someMember"];
[foo valueForKeyPath:@"someMember.someAttributeOnMember"];
[foo valueForKeyPath:@"someOtherMember"];

//无效
[foo valueForKeyPath:@"4fun"];
[foo valueForKeyPath:@"kermit the frog"];
[foo valueForKeyPath:@"SomethingWickedThisWayComes"];
[foo valueForKeyPath:@"THISWONTWORK"];
[foo valueForKeyPath:@"thisAlsoWon'tWork"];
```

前面介绍过，可以通过键路径来遍历对象间的关系并访问子对象的特性。键路径是一个键，其中不同对象之间的关系通过点操作符隔开，如代码清单 6-5 中的`@"someMember.someAttributeOnMember"`。键路径访问 `foo` 对象的 `someMember` 特性，在 `someMember` 所属的类上查找 `someAttributeOnMember` 特性，并返回存储在这里的值。在键路径的使用规范中提供了很多语法糖。除了可以遍历关系外，还可以访问一些操作对象集合的函数，如计数等。比如，代码清单 6-6 就显示了可以在键路径中使用的一些内置函数。

代码清单 6-6　在键路径中使用函数

```
[anArrayOfProducts valueForKeyPath:@"@avg.price"];
[anArrayOfProducts valueForKeyPath:@"@sum.cost"];
[store valueForKeyPath:@"products.@count"];
```

函数只能在对象数组和对象集合上使用。在本例中，前两行代码访问产品对象集。这些产品有价格和成本等特性。给定的函数接收数组中每个对象的指定特性值，在这些值上调用指定的函数。换句话说，第一行代码的作用是遍历产品数组中的每个元素，收集其中的每个对象的价格属性，然后求平均值。

使用这些函数的语法是，以@前缀开头，后接函数名，一个“.”符号以及要操作的特性。`@count` 函数是个例外，不必为其指定特性，因为该函数只是简单返回集合中的元素个数。

表 6-1 显示了这些函数的列表。

表 6-1 内置函数

函 数	功 能
@avg	返回数组或者集合中所有元素的平均值
@count	返回数组或者集合中元素的个数
@max	返回数组或者集合中所有元素的最大值
@min	返回数组或者集合中所有元素的最小值
@sum	返回数组或者集合中所有元素的总和
@unionOfArrays/@distinctUnionOfArrays	给定一系列数组，返回一个包含所有数组的数组。对于 distinct 版本，返回数组中的元素不重复
@unionOfSets/@distinctUnionOfSets	给定一系列集合，返回一个包含所有集合的集合。对于 distinct 版本，返回集合中的元素不重复
@unionOfObjects/@distinctUnionOfObjects	给定一系列集合或者数组，返回一个包含所有元素的数组。对于 distinct 版本，返回数组中的元素不重复

6.1.2 编写符合KVC标准的存取器方法

Objective-C 运行时之所以能实现很多神奇的功能，一部分要归功于底层框架的能力，另外则要归功于编码风格约定。尽管 Objective-C 运行时对于 KVC 功能做了大量工作，但还是需要开发者在编写属性的存取器方法时遵循特定的规则，这样它才可以通过 KVC 获取并且设置值。

赋值方法和取值方法的 KVC 标准遵循以下模式：赋值方法是 `set<Value>:`，而取值方法就是简单的`<value>`。在这两种情况下，`<value>`部分都必须替换成你要访问的属性名。该属性必须是名字符合驼峰命名法的成员变量。



说明

驼峰命名法指的是一种变量命名标准，它将变量名中的单词拼装成长字符串。在该字符串中，除了第一个单词外每个单词的首字母都是大写的。比如，如果要使用 some variable name 作为变量名，就应该将其命名为 someVariableName。这种变量名中的大写字母看起来有点像骆驼的驼峰，这也是其名字的由来。

代码清单 6-7 给出了类定义和存取器方法的代码。

代码清单 6-7 带有成员变量的类定义

```
@interface MyClass : NSObject
{
    float x;
    float y;
    NSString *something;continued
```

```
}
-(void)setX:(float)inX;
-(void)setY:(float)inY;
-(void)setSomething:(NSString *)inSomething;
-(float)x;
-(float)y;
-(NSString *)something;
@end
//此类内容的列表

//符合 KVC 标准的存取器方法

@implementation MyClass
-(void)setX:(float)inX;
{
    x = inX;
}

-(void)setY:(float)inY;
{
    y = inY;
}

-(void)setSomething:(NSString *)inSomething;
{
    NSString *oldValue = something;
    something = [inSomething retain];
    [oldValue release];
}

-(float)x;
{
    return x;
}

-(float)y;
{
    return y;
}

-(NSString *)something;
{
    return something;
}

@end
```

如果使用属性（property）来封装了对象的特性（attribute），就会自动生成符合 KVC 标准的存取器方法。换句话说，如果使用属性，就不需要做其他处理了。也就是说，之前的说明是适用的。但如果在使用属性的同时重写了标准的属性存取器方法的命名规则，存取器方法就不符合 KVC 标准，从而无法使用 KVC。

6.1.3 在数组中使用KVC

在考虑面向对象的设计时，最好应该考虑对象之间的关系。当一个类的成员变量的类型是另一个类型时，这种关系就是所谓的一对一的关系。代码清单 6-8 显示了一个该类型的类接口示例。

代码清单 6-8　有一对一关系的 Foo 类和 Bar 类

```
@interface Foo : NSObject
{
    Bar *bar;
}
@property (retain) Bar * bar;
@end;
```


比如，如果给定的类包含一成员变量，而该变量实际上是其他对象的集合，这种关系就是一对多的关系。一对多的关系通常通过 NSArray 或者 NSSet 成员变量来实现，成员变量中包含的元素就是另一个类的实例。

代码清单 6-9 就显示了这类关系的示例。需要注意的是，数组包含 Bar 类的实例，但没有在这里指定。

代码清单 6-9　有一对多关系的 Foo 类和 Bar 类

```
@interface Foo : NSObject
{
    NSArray *bars;
}
@property (retain) NSArray * bars;
@end;
```

在这些情况下通过 KVC 访问这些值时，你可能不是想访问数组成员变量，而是想直接访问数组中的元素。在某些情况下，这可能会更高效，至少它更直接地表示了两个对象之间存在的实际关系。

KVC 提供了一个用于此类操作的特殊的存取器方法集合。它们专门用于访问一些一对多关系的属性以及在这些关系中所涉及的每个元素。

这些存取器方法有两大类：第一类是索引存取器方法。这些存取器方法可用于访问数组所表示的一对多关系中的每个元素。NSArray 就是有序集合，这种情况下经常使用这种容器。一对多关系中第二类存取器方法适用于关系成员变量位于无序集合中的情况，比如 NSSet。在这种情况下，用于访问这类一对多关系中的元素的存取器方法也称作无序存取器方法。这两种类型的存取器方法都有可变和不可变版本。



说明

尽管这类关系通常都通过 NSArray 和 NSSet 来体现，但从技术上来说可以通过任何所选的集合来建模。关键点就是所创建的存取器方法必须符合每种类型的访问所对应的约定。

1. 使用索引存取器方法

在一个一对多的关系中使用索引存取器方法时，必须实现获取元素总数的方法 `-countOf<VariableName>`，以及为了获取这种关系的元素必须选择的其他一些方法。用于在索引集合中获取元素的方法是 `-object<VariableName>AtIndex:` 或者 `-<variableName>AtIndexes:`。设计这些方法的目的就是让类的用户可以访问一个或多个给定索引对应的元素，并应该返回相应的对象。这些存取器方法的实现示例包括代码清单 6-10 所示的从 `Foo` 到 `Bar` 的一对多关系。

代码清单 6-10　不可变的一对多索引存取器方法的实现

```
@implementation Foo

-(NSUInteger)countOfBars
{
    return [bars count];
}

-(id)objectInBarsAtIndex:(NSUInteger)inIndex;
{
    return [bars objectAtIndex:inIndex];
}

//或者

-(NSArray *)barsAtIndexes:(NSIndexSet *)inIndexes;
{
    return [bars objectsAtIndexes:inIndexes];
}

//或者

-(void)getBars:(Bar **)outBuffer range:(NSRange)inRange;
{
    [bars getObjects:outBuffer range:inRange];
}

@end
```

除了这些存取器方法外，你还可以实现可选方法 `-get<VariableName>:range:`，该方法通过限制在代码清单 6-10 所示数组中的指定范围内进行搜索，从而实现更好的性能。结果会存储在 `outBuffer` 变量中。

你可以想象如果在之前实现 `Foo` 和 `Bar` 之间的关系时使用的是 `NSMutableArray`，这种关系就成为了可变的一对多关系。如果需要实现可变的一对多关系，也就是可以增加、删除和更改索引集合中的元素，就必须实现 `-insertObject:in<VariableName>AtIndex:` 或 `-insert<VariableName>:atIndexes:` 来插入元素，`-removeObjectFrom<VariableName>AtIndex:` 或 `-remove<VariableName>AtIndexes:` 来移除元素，以及用于高性能的对象替换的 `-repla-`

ceObjectIn<VariableName>AtIndex:withObject:或-replace<VariableName>AtIndexes:with<VariableName>:。替换对象是一种只在衡量性能的情况下的可选操作。通常，替换给定索引处的对象，而不是移除原对象并重新插入一个对象，会是一种更快的方法。是否实现这些方法由你决定。

在该关系中实现 Foo 和 Bar 之间可变访问所需的其他代码如代码清单 6-11 所示。

代码清单 6-11　通过一个索引集合实现可变的一对多关系

```
-(void)insertObject:(Bar *)inBar inBarsAtIndex:(NSUInteger)inIndex;
{
    [bars insertObject:inBar atIndex:inIndexes];
}

-(void)insertBars:(NSArray *)inBars atIndexes:(NSIndexSet *)inIndexSet;
{
    [bars insertObjects:inBars atIndexes:inIndexSet];
}

-(void)removeObjectFromBarsAtIndex:(NSUInteger)inIndex;
{
    [bars removeObject:inIndex];
}

-(void)removeBarsAtIndexes:(NSIndexSet *)inIndexSet;
{
    [bars removeObjectsAtIndexes:inIndexSet];
}

-(void)replaceObjectInBarsAtIndex:(NSUInteger)inIndex
                       withObject:(id)inBar;
{
    [bars replaceObjectAtIndex:inIndex withObject:inBar];
}

-(void)replaceBarsAtIndexes:(NSIndexSet *)inIndexSet
                  withBars:(NSArray *)inBars;
{
    [bars replaceObjectsAtIndexes:inIndexSet withObjects:inBars];
}
```

2. 使用无序存取器方法

处理使用的对象集合是无序集合的一对多关系时，要实现一套不同的符合 KVC 标准的存取器方法。

和处理索引存取器方法一样，存在可以从集合读取值的不可变存取器方法，以及用来改变集合中的值的可变存取器方法。

对于不可变的存取器方法，和索引集合一样，你必须实现-countOf<VariableName>方法来返回集合中的元素个数。此外，你还必须实现-enumeratorOf<VariableName>和-memberOf<VariableName>:方法。对于-enumeratorOf<VariableName>，该方法返回的是一个经过初

始化的 NSEnumerator 对象，用于遍历该集合。对于-memberOf<VariableName>:，该方法接收一个对象实例作为参数，并返回集合中和该对象 isEqual:比较为真的所有对象。如果集合内任何对象的 isEqual:结果都不为真，则它就应该返回 nil。

代码清单 6-12 显示了使用 NSSet 的一对多关系中这些方法的实现示例。

代码清单 6-12　不可变无序集合中的存取器方法

```
@implementation Foo

-(NSUInteger)countOfBars
{
    return [bars count];
}

-(NSEnumerator *)enumeratorOfBars
{
    return [bars objectEnumerator];
}

-(Bar *)memberOfBars:(Bar *)inBar;
{
    return [bars member:inBar];
}

@end
```

如果要改变无序的一对多关系，就需要实现-add<VariableName>Object:或-add<VariableName>:方法来加入新的对象，-remove<VariableName>Object:或-remove<VariableName>:来移除对象，以及-intersect<VariableName>:来移除集合中的一组对象。

代码清单 6-13 显示了通过 NSSet 来实现这种关系时这些方法的实现示例。

代码清单 6-13　无序一对多关系中可变存取器方法的实现

```
//添加

-(void)addBarsObject:(Bar *)inBar;
{
    [bars addObject:inBar];
}

//或者

-(void)addBars:(NSSet *)inBars;
{
    [bars unionSet:inBars];
}

//移除

-(void)removeBarsObject:(Bar *)inBar;
```

```
{
    [bars removeObject:inBar];
}

//或者

-(void)removeBars:(NSSet *)inBars;
{
    [bars minusSet:inBars];
}

//交叉

-(void)intersectBars:(NSSet *)inBars;
{
    return [bars intersectSet:inBars];
}
```


6.1.4 在结构体和标量中使用KVC

使用KVC时的一个重要限制就是其所有的方法，`-valueForKey:`、`-setValueForKey:`等，输入参数和返回值都是`id`数据类型的。对于大多数特性，这不会是问题，因为大多数情况下是通过对象来定义这些特性的，因此可以通过`id`数据类型来控制这些特性。但是，当有一些结构体或者标量类型的特性时，比如`int`、`float`等，就会造成一些问题。

具体来说，Objective-C运行时环境在使用键值编码时实际不能直接使用这些类型的变量。它必须将这些值从原类型转换成完全的Objective-C对象。

所幸，大多数情况下Objective-C都可以为你处理这些。如果你要存取器的KVC特性不是对象，Objective-C运行时就会自动并透明地查看要访问的变量类型，并创建一个`NSNumber`和`NSValue`对象来包装该值，使得它可以用于KVC。

在本书的第三部分我会介绍`NSNumbers`和`NSValues`，不过目前只需要知道它们是用来包装标量和构造体，并将其作为Objective-C对象的特殊类。这样你就可以在数组、字典中存储它们并在KVC中使用它们。

再次指出，Objective-C为你自动处理这些，不过有一个需要注意的特殊情况。如果在使用KVC存取器方法来设置一个标量值时传入`nil`值，就不存在能在所有情况下自动处理这一事件的通用方法。结果就是在这一情况下，Objective-C运行时会调用`-setNilValueForKey:`方法。该方法在默认情况下会抛出异常。如果需要，你就可以在类中重写该方法来进行任何适当的处理。比如，你可以定义为类的某个特定变量传入`nil`值就意味着该值必须是–1.0。在这种情况下，你将重写`-setNilValueForKey:`方法，检查传入的键是什么，如果所定义的变量在`nil`的情况下应该是–1.0，你就可以自己创建一个`NSNumber`实例，并通过`-setValue:forKey:`方法手动设置该值。

再次指出，这仅仅是一种特殊情况。大多数情况下，将标量和结构体包装到`NSNumbers`和`NSValues`中以及从其中解包的过程是完全透明的。即使是自定义的结构体也可以通过`NSValue`的`-getValue:`自动处理。

6.1.5　查找对象特性

在访问符合 KVC 标准的特性时，运行时在查找给定键路径对应的存取器方法时会遵循一套特定的规则。这些规则如下。

在设置一个特定键对应的值时，运行时首先会在类上搜索特定的存取器方法，这些方法遵循之前提到的标准存取器方法模式，即`-set<ValueName>:`存取器方法。如果没有找到该存取器方法，你的类还可以实现可选方法`-accessInstanceVariablesDirectly`并返回`YES`。在本例中，运行时就会按照以下顺序在类上查询任何遵循`_<valueName>`、`_is<valueName>`、`<valueName>`或者`is <valueName>`等命名模式的实例变量。如果没有任何一个符合，就会调用`-setValue:forUndefinedKey:`。该方法的默认行为就是引发一个异常。

在使用键值编码获取一个值时，运行时也按照一个类似的流程来查找一个给定键所表示的变量。具体来说，首先会在类上按如下顺序查找名字符合`-get<ValueName>`、`-<valueName>`或`-is<ValueName>`模式的存取器方法。如果找到此类存取器方法，就会通过它获取该值。如果没有找到符合这些规格的存取器方法，就会尝试确定要存取器的值是否为数组。为此就会检查符合`-countOf<ValueName>`、`-objectIn<ValueName>AtIndex:`和`-<valueName>AtIndexes:`模式的方法。这些数组 KVC 存取器方法的存在表明，所访问的值是一个存储在成员变量中的数组。如果找到这些存取器方法，运行时就会返回一个代理对象`NSArray`，该对象包含它所找到的和上述存取器方法相关的代理方法。访问该`NSArray`对象的其中任何一个方法都会调用原对象的相应存取器方法。

之后，运行时会确定你所访问的值是否可以作为集合访问到。为此，它会检查`-countOf<ValueName>`、`-enumeratorOf<ValueName>`、`-memberOf<ValueName>:`等方法。如果所有的方法都找到，就会返回一个代理对象`NSSet`。访问该代理对象时，如果对该代理对象调用上述方法中的任何一个，就会自动调用原对象的相应方法。

再次指出，和设置值类似，如果实现了类方法`-accessInstanceVariablesDirectly`并返回`YES`，运行时就会查找符合`-_<valueName>`、`-_is<ValueName>`、`-<valueName>`或者`-is<ValueName>`这类标准命名规则的成员变量。此外，和赋值方法一样，如果它发现任何这类成员变量，就会直接访问。

最后，如果这些都不适用，运行时就和设置值一样调用方法`-valueForUndefinedKey:`。

6.2　观察对符合 KVC 标准的值的修改

基于键值编码的一种简洁的 Objective-C 技术就是键值观察。利用键值观察你可以注册成为一个给定对象的观察者，并在该对象的某个属性变化时收到通知。这是极其强大的功能，并且被内置为 Objective-C 的核心部分。

编写 KVC 存取器方法似乎需要很多努力（尽管在使用属性时并不需要），但是为所有类特性创建符合 KVC 标准的存取器方法的好处就是可以免费获得键值观察。

6.2.1　使用KVO

利用键值观察（Key Value Observing），你可以自动观察其他对象的变化。因此，当一个对象改变状态时你就会得到通知，比如用户通过应用中的设置面板改变了设置时。通过键值观察，利用该设置的窗体和其他对象在用户改变该设置时，都可以自动得到通知。你不需要手动告诉其他对象进行更新。它们会自动收到新值并执行适当的操作。这极其强大。设置是该技术最强大的应用之一，此外，Cocoa 框架中的核心数据和其他技术也利用了键值观察实现了一些奇妙的功能。

要使用键值观察，被观察的对象必须对所观察的特性使用符合 KVC 标准的存取器方法。第二，想要观察变化的对象，也就是观察者，必须实现一个接收通知的特殊方法。该方法是 `-observeValue:forKeyPath:ofObject:change:context:`。该方法在值变化时被调用并可以配置成同时接收新值和原值以及其他自定义的信息。

最后，观察者通过调用`-addObserver:forKeyPath:options:context:`方法针对被观察对象进行注册。调用该方法，告诉对象要观察的 KVC 键路径以及希望看到的变化，并提供一个在收到变化通知时可以传回的上下文对象。

但观察者完成这些配置后，键路径指定的属性的任何变化都可以自动调用观察者的回调方法。在观察者完成对被观察对象的观察后，必须将自己移除。如果没有做到这点并且观察者之后就释放了，将来向该观察者发送通知时可能会导致应用崩溃。

6

6.2.2　注册成为观察者

注册成为观察者很容易。针对想要观察的对象简单调用`-addObserver:forKeyPath:options:context:`方法，如代码清单 6-14 所示。

代码清单 6-14　增加一个观察者

```
[obj addObserver:self
      forKeyPath:@"memberVariable"
         options:(NSKeyValueObservingOptionNew |
                  NSKeyValueObservingOptionOld)
         context:NULL];
```

`Observer` 参数通常是 `self`，这是在被观察值变化时收到通知的对象。键路径参数指定想要观察的特性的键路径。`options` 参数指定一些标记来告诉 KVO 你希望变化如何传给你。这些值可以通过 | 操作符进行或操作。传入的可能值如表 6-2 所示。

表 6-2　传入的可能值

值	功　　能
`NSKeyValueObservingOptionNew`	作为变更信息的一部分发送新值
`NSKeyValueObservingOptionOld`	作为变更信息的一部分发送旧值
`NSKeyValueObservingOptionInitial`	在观察者注册时发送一个初始更新
`NSKeyValueObservingOptionPrior`	在变更前后分别发送变更，而不只在变更后发送一次

上下文参数是一个在 KVO 系统中无变更传递的 void*参数，并且会在有变更通知时传回给你的对象。本质上，就 KVO 而言，该参数是不透明的数据块，并且完全是和实现相关的。任何从此传入参数的都会无变更传递的。



说明

记住使用 void*上下文参数时有和垃圾回收相关的特殊规则，你必须确保 void*指向的数据在之后访问时仍然没有被释放并有效。换句话说，不要将一些存储在栈上的值传递给该参数。这会导致崩溃。

在注册成为观察者之后，如果传入 NSKeyValueObservingOptionInitial 标志，你会得到一个所观察特性的初始值的最初通知。此外，每次值变化时，你都可以收到这些变化的通知。

为了收到这些通知，必须实现下节要介绍的回调方法。

6.2.3　定义KVO的回调

使用 KVO 的第二步就是编写观察者的回调方法。代码清单 6-15 显示了-observeValue:forKeyPath:ofObject:change:context:方法的一个示例实现。

代码清单 6-15　KVO 回调方法的实现示例

```
-(void)observeValueForKeyPath:(NSString *)inKeyPath
                     ofObject:(id)inObject
                       change:(NSDictionary *)inChange
                      context:(void *)inCtx;
{
    if([inKeyPath isEqualToString:@"memberVariable"])
    {
        NSString *newValue = [inChange
               objectForKey:NSKeyValueChangeNewKey];
       //对新值进行一些处理
    }
    else if([inKeyPath isEqualToString:@"..."])
    {

    }
    [super observeValueForKeyPath:inKeyPath
                         ofObject:inObject
                           change:inChange
                          context:inCtx];

}
```

可以从该方法看出，要做的第一件事情就是找出被观察对象中变化的特性。该方法自动传入一个对象参数，告诉你哪个对象向你发送通知。通过对键路径的传入值使用-isEquals 方法，你可以准确地确定对象的什么特性发生了改变。Key 参数仅仅是一个字符串，和对 KVC 使用时

一样。因此，可以使用 NSString 方法-isEqualToString:来确定该通知所对应的键路径。

在确定了对象的哪个特性发生变化后，你可以执行任何合适的操作。实际的变化通过 change 参数传递给你。该参数是一个 NSDictionary 对象，包括和你注册成为观察者时所请求的变化信息相关的键和值。这些键和值如表 6-3 所示。

表 6-3　和变化信息相关的键值

键	值
NSKeyValueChangeKindKey	指定变化类型的 NSNumber
NSKeyValueChangeNewKey	新值
NSKeyValueChangeOldKey	原值
NSKeyValueChangeIndexeskey	如果 NSKeyValueChangeKindKey 是 NSKeyValueChangeInsertion、NSKeyValueChangeRemoval、NSKeyValueChangeReplacement 之一，该值就包含变化值的索引
NSKeyValueChangeNotificationIsPriorKey	和 NSKeyValueChangeOptionPrior 结合使用来表示这是“之前”的通知

可以看出，如果选择同时接收原值和新值，两个都会在变化参数中提供，通过合适的键就可以访问到。从变化字典中获取到值之后，就可以在对象中使用它执行任何需要的操作。

记住 KVC 必须使用对象来发送值——不能直接使用标量和结构体。因此，如果所观察的值是标量或者结构体，所接收的值就分别是 NSNumber 或 NSValue 类型的。因此，必须从该值中提取出实际需要的标量或者结构体值。上述示例代码就展示了这一点。

NSKeyValueChangeKindKey 指定了接收到的变化类型。可能的类型如表 6-4 所示。

表 6-4　可能的变化类型

值	功　　能
NSKeyValueChangeSetting	指定该值被设置
NSKeyValueChangeInsertion	指定这些值插入到集合或者一对多的关系中
NSKeyValueChangeRemoval	指定这些值在一对多的关系中被移除
NSKeyValueChangeReplacement	指定这些值在一对多的关系中被替换

6.2.4　移除观察者

记住在结束对一个对象变化的观察后，需要移除观察者。如果不这样，应用可能会崩溃。



说明

在垃圾回收的环境中，如果忘记移除观察者可能不会造成崩溃。但是，移除观察者仍是一种好的做法，这样就可以在不支持垃圾回收的环境中形成该习惯。

为了移除观察者，只需要调用方法-removeObserver:forKeyPath:，并传入观察者作为第一个参数，观察的键路径是第二个参数。代码清单 6-16 显示了一个在观察者的 dealloc 方法中实现的示例。

代码清单 6-16　移除观察者

```
-(void)dealloc;
{
    [obj removeObserver:self forKeyPath:@"memberVariable"];
    [super dealloc];
}
```

6.2.5　实现手动通知

所有的这些通知都自动发生。需要做的就是为属性提供符合 KVC 标准的存取器方法，其他一切都会正常工作。有时，不一定想利用自动通知。有时想在改变一个值或者一组值时手动发送通知。比如，如果需要一次性进行很多变更，可能会想将这些变化打包后一起发送。在这些情况下就会使用手动通知。

为了实现手动通知，必须重写类方法+automaticallyNotifiesObserversForKey:来告诉 Objective-C 你不想自动通知观察者所发生的变化。可以通过对任意一个想进行手动通知的键返回 NO 来实现。示例如代码清单 6-17 所示。

代码清单 6-17　重写+automaticallyNotifiesObserversForKey:

```
+(BOOL)automaticallyNotifiesObserversForKey:(NSString *)inKey;
{
    if([inKey isEqualToString:@"memberVariable"])
        return NO;
    return YES;
}
```

如果想要手动通知所发生的变化，你必须在变化之前调用方法-willChangeValueForKey:，然后在变化之后调用方法-didChangeValueForKey:。示例如代码清单 6-18 所示。

代码清单 6-18　实现手动通知

```
-(void)setMemberVariable:(CGFloat)inValue;
{
    [self willChangeValueForKey:@"memberVariable"];
    memberVariable = inValue;
    [self didChangeValueForKey:@"memberVariable"];
}
```

这些调用在需要时是可以嵌套的，比如在一次调用中需要修改多个变量的情况。此外还有和一对多关系对应的调用。它们是-willChange:valuesAtIndexes:forKey:和-didChange:valuesForIndexes:forKey:。

6.2.6 使用KVO的风险

使用 KVO 也会遇到问题。想让计算机自己“做事情”的时候，就有可能会因一些不平常的因素组合而导致问题。KVO 也一样。

更具体点说，使用 KVO 最大的风险就是如果观察者观察每一步，这些观察者可能会执行其他操作，因为你无法控制这些观察者，所以也就无法控制这些操作。

大多数情况下这不会成为一个问题，但也在例外。这种情况就是在初始化函数或者 dealloc 函数中使用存取器方法来释放变量成员时，如代码清单 6-19 所示。

代码清单 6-19　在 dealloc 中使用存取器方法释放成员变量

```
-(void)dealloc
{
    [self setFoo:nil];
    [self setBar:nil];
    [super dealloc];
}
```

按这种方式写 dealloc 方法很好！可以在释放成员变量的同时将它设成 nil，一步搞定。

问题就是，在调用这些存取器方法时，KVO 观察者会在这些变化发生时收到通知。如果他们不想接收 nil 或者希望在收到通知时能够处理对象本身，此时就会发生一些糟糕的事情。此外，如果想到了观察者在收到 bar 变量的变化通知时，希望可以访问 foo 变量，这种情况下就会造成一个问题，因为 foo 变量已经被释放并且被设置成 nil。

苹果目前推荐的做法就是不要在初始化函数或者 dealloc 方法中通过存取器方法初始化和释放成员变量。这种情况在 64 位运行时中变得更复杂，因为它可以在没有相关的成员变量的情况下声明属性。在这些情况中，初始化和释放成员变量的唯一办法就是通过存取器方法。

我是使用存取器方法来初始化和释放成员变量的，除非我知道在给定的环境中这样做会造成问题。此外，实现键值观察者时，我会确保观测者可以正确处理 nil 值并尽量最小化副作用。

如果你觉得这种风险是值得的，那就通过存取器方法来编写初始化函数和释放函数吧。只要意识到可能的危险，在遇到问题时，就可以马上知道应该从哪里查找。

另一方面，如果你不能确保观察者会这么做的话，那就就遵循苹果的建议，除非不得已，否则不要在初始化函数和析构函数中使用存取器方法。

6.3 应用键值观察

现在你可能知道了键值编码和键值观察的细节，看看由几个类构成的一个小示例应用。其中的一个类会观察其他类，在收到被观察类的特性变化通知时，会在控制台输出那些变化。

代码清单 6-20 显示了应用中的第一个类。这个简单的点类有两个属性，x 和 y。将如下接口放在接口文件，并将实现放在实现文件。

代码清单 6-20　MyPoint 类的接口和实现

```
//接口，将其放在 MyPoint.h 文件中
#import <Cocoa/Cocoa.h>

@interface MyPoint : NSObject
{
    CGFloat x;
    CGFloat y;
}
@property CGFloat x;
@property CGFloat y;

@end
//实现，将其放在 MyPoint.m 文件中
#import "MyPoint.h"

@implementation MyPoint
@synthesize x;
@synthesize y;

@end
```

代码清单 6-21 显示了 Observer 类。该类接收一个 MyPoint，并将自己加为观察者。本身没有任何东西。

代码清单 6-21　Observer 类

```
//接口，将其放在 Observer.h 文件中
#import <Cocoa/Cocoa.h>
#import "MyPoint.h"

@interface Observer : NSObject
{
    MyPoint *point;
}
@property (retain) MyPoint *point;
-(id)initWithPoint:(MyPoint *)inPoint;
@end

//实现，将其放在 Observer.m 文件中
#import "Observer.h"

@implementation Observer
@synthesize point;

-(void)dealloc;
{
    [point removeObserver:self forKeyPath:@"x"];
    [point removeObserver:self forKeyPath:@"y"];
    [point release];
    point = nil;
    [super dealloc];
```

```
}

-(id)initWithPoint:(MyPoint *)inPoint;
{
    if(self = [super init])
    {
        point = [inPoint retain];
        [point addObserver:self forKeyPath:@"x"
                   options:(NSKeyValueObservingOptionNew|
                            NSKeyValueObservingOptionOld)
                   context:nil];
        [point addObserver:self forKeyPath:@"y"
                   options:(NSKeyValueObservingOptionNew|
                            NSKeyValueObservingOptionOld)
                   context:nil];
    }
    return self;
}

-(void)observeValueForKeyPath:(NSString *)keyPath
                     ofObject:(id)object
                       change:(NSDictionary *)change
                      context:(void *)context;
{
    NSNumber *oldValue = [change objectForKey:NSKeyValueChangeOldKey];
    NSNumber *newValue = [change objectForKey:NSKeyValueChangeNewKey];


    if(keyPath == @"x")
        NSLog(@"Value for X changed from: %f to %f",
              [oldValue floatValue],
              [newValue floatValue]);
    if(keyPath == @"y")
        NSLog(@"Value for Y changed from: %f to %f",
              [oldValue floatValue],
              [newValue floatValue]);
}

@end
```

代码清单 6-22 给出了主函数的代码。

代码清单 6-22 主函数

```
#import <Foundation/Foundation.h>
#import "Observer.h"
#import "MyPoint.h"

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    MyPoint *point = [[MyPoint alloc] init];
    Observer *observer = [[Observer alloc] initWithPoint:point];
```

```
    point.x = 42.0;
    point.y = 55.1;

    point.x = 4200.0;
    point.y = 5500.1;

    [observer release];
    [point release];

    [pool drain];
    return 0;
}
```

它所做的就是创建一个点，然后创建一个观察者，同时传入该点。之后改变点的位置。所有的输出都发生在 `Observer` 类内部，并且都是自动的。

编译并运行该应用来验证我所说的。

6.4 小结

本章介绍了键值编码和键值观察这两个 Objective-C 以及 Cocoa 和 Cocoa Touch 框架提供的核心技术。通过该功能，你可以构建更灵活的应用设计以使原本紧密耦合的应用的不同部分可以松散耦合。松散耦合意味着更灵活的设计，KVO 和 KVC 提供的工具可以让应用的组件松散耦合。

第 7 章

使用协议

7

本章概要

- ❑ 通过协议解决面向对象设计问题
- ❑ 在类中实现协议
- ❑ 采用协议
- ❑ 可选方法的使用
- ❑ 理解正式协议和非正式协议的区别

Objective-C 不支持多继承，这有利也有弊。优势在于多继承会导致很难处理的复杂问题，而劣势就是有时你想创建一个类来实现一个特定的接口，而不想从定义该接口的类继承。幸运的是，Objective-C 的设计者们加入了一个可以处理这种情况的功能，即协议。

本质上，协议就是其他多个类不通过继承就可实现的接口。这样你就可以混合并匹配给定类的功能，从而使该类可以适用不同的使用场景。

7.1 优先使用组合而不是继承

“优先使用组合而不是继承”这一个面向对象的设计原则意味着，在扩展给定类的功能时不能一味地使用继承，而应该首先尝试通过在类中组合其他类来解决问题。比如，如果需要实现一个在网络服务和应用之间提供接口的类，应该在类中包含另外一个提供网络连接性的对象，而不是从 `socket` 类（一个用于访问网络资源的类）继承。你可以利用其他可用组件“组合”你的设计。通过这种方式构建的设计会更灵活，因为组合在一起的每个部分今后都可以替换以解决其他问题。这是一个极其强大的设计思想，在代码中应该努力做到这一点。

通过使用标准的面向对象技术显然可以遵循这种设计理念。但是，设计可复用的组件会增加复杂性。

图 7-1 显示了一个类图，用来表示业务逻辑类和一个提供网络连接性的类之间的关系。这是一个经典的“组合”设计。

设计思路就是 `NetworkConnector` 类提供了所有和网络服务器的交互，包括连接、断开连接和收发数据。`BusinessLogic` 类会接收 `NetworkConnector` 收到的数据，并确定应用中数据的走向。

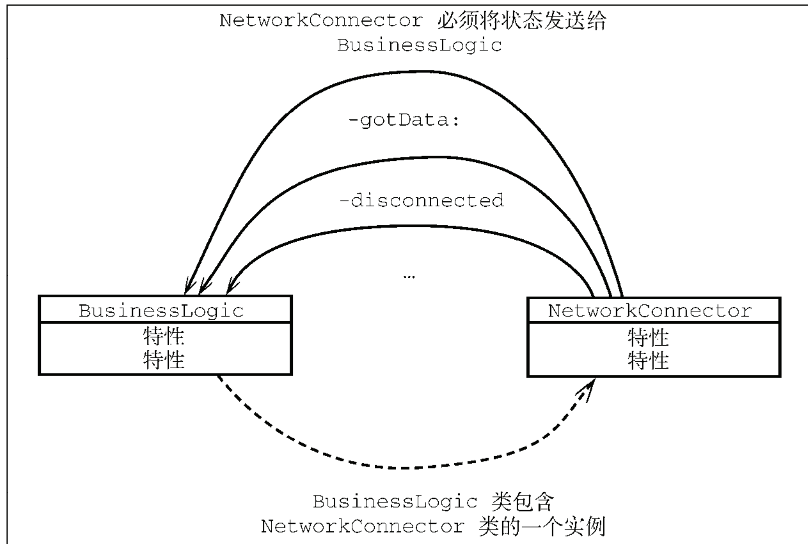


图 7-1　紧密耦合的 BusinessLogic 和 NetworkConnector

接收数据时，如果 NetworkConnector 类需要通知 BusinessLogic 类就会有问题。网络连接性是一个很通用的概念，这是一个在其他应用或者同一个应用的其他地方一定要复用的东西。如果想将网络类设计成一个完全通用的、可以不断复用的类，就必须将其设计成和客户端类（即本例中的 BusinessLogic 类）没有紧密耦合。

NetworkConnector 不能依赖 BusinessLogic 类,因为你不能指望着使用 NetworkConnector 类的都是 BusinessLogic 类。

一个解决的办法就是使用继承来强制 BusinessLogic 类继承自 NetworkConnector 类可以依赖的类。该方案如图 7-2 所示。

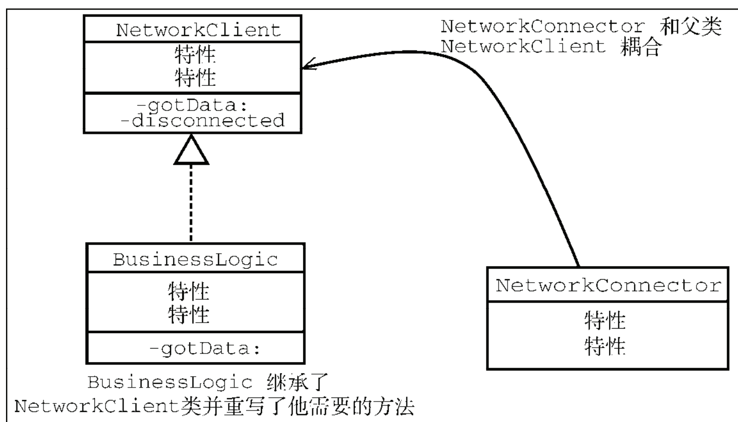


图 7-2　BusinessLogic 类从通用的 NetworkClient 类继承

这样，客户类的开发人员必须被迫从一个特定的父类继承。如果该父类不能被流畅地加入到开发者的对象继承中，就会造成很大的设计问题。比如，试想一下 BusinessLogic 类需要同时与网络和硬盘的 I/O 系统通信，就需要有一个相似的回调机制。像这样直接继承就无法处理这种情况。

7.1.1 了解为什么不需要（或不想要）多继承

C++等一些语言通过多继承解决这类问题。图 7-3 显示了一个在 C++中如何解决网络通信问题的示例。

多继承的问题通常称作“钻石问题”，如图 7-4 所示。

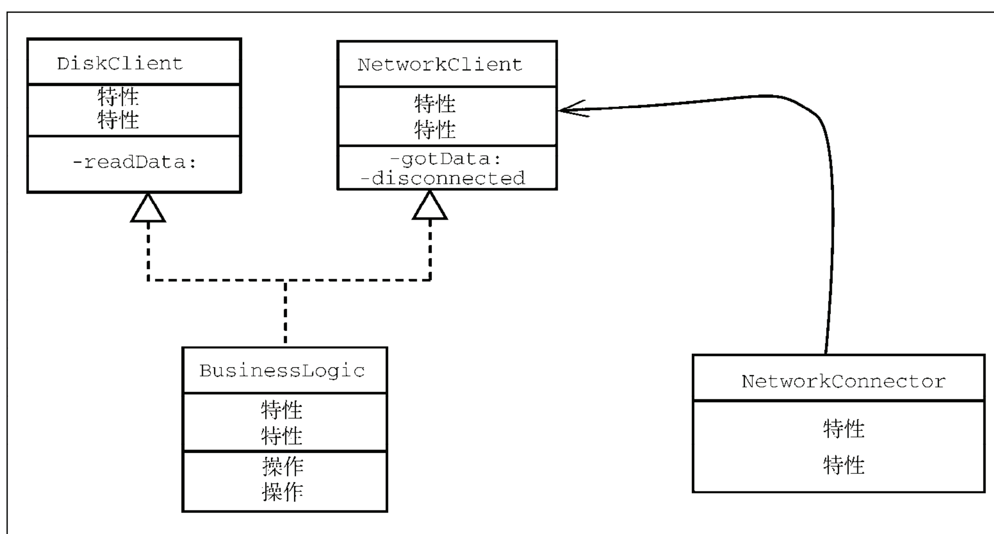


图 7-3 使用多继承

从类图中可以看出，多继承的主要问题出现在同时继承两个不同的类（在图 7-3 中，类 D 同时从类 B 和 C 继承）的情况下，同时它们也继承自共同的超类（类 A）。在这种情况下，如果在类 D 的实例上调用类 A 的方法就会发生二义性。这种情况下，如果类 D 没有重写该方法并提供自己的实现，哪个父类的方法会被调用呢？B 还是 C？

因此，Objective-C 不提供多继承。如果只有单继承，就无需担心“钻石问题”了。

7.1.2 理解协议如何解决问题

协议支持声明一个接口来解决该问题，这个接口是在不提供默认实现时由一个类实现的。协议不提供一种在其中指定方法实现的机制。它们只提供用于声明这些方法的接口的机制。可以让可复用组件不依赖于特定的类实现，而是依赖于以协议形式存在的接口。实现了给定协议的类就要提供协议声明中指定的方法的实现。通过这种方式实现协议类就可以仅仅依赖于接口，并且由

于声明实现给定接口的类必须同时要实现相应的方法，于是就避免了二义性问题。也不会有“钻石问题”，因为任何声明和支持的协议在类中必须有一个实现。

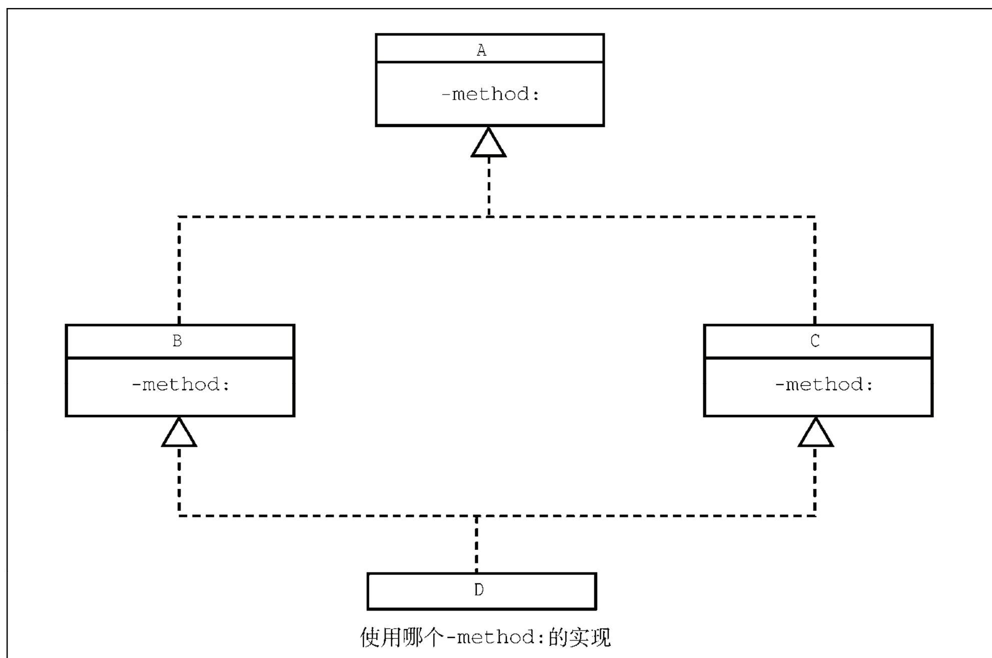


图 7-4　钻石问题

从面向对象设计的角度来看，该解决方案的类图和多继承的类图几乎一致，但是它不存在多继承中不可避免的问题。

该解决方案不是 Objective-C 独有的。很多方面都基于 Objective-C 的 Java 也实现了一个类似的概念，称作接口，如果有 Java 背景，协议和接口的概念应该很类似，你应该会感觉很熟悉。

7.1.3　记录期望别人实现的接口

协议的另一种理解方式就是将其想象成记录在文档中需要其他人实现的接口。在 `NetworkConnector` 一例中，我们记录了所有网络类需要开发者协助确定如何处理的不同情况。

比如如果 `NetworkConnector` 从网络接收数据，应该很自然认为应用本身应该有确定如何处理这些数据所需的知识（业务逻辑）。换句话说，没有一种通用的方式说“当我接收到数据，我就这样处理”。你必须问一下应用的剩余部分“我刚收到数据，你想让我如何处理呢？”，这是特别适合使用协议的情况。你可以为不同的未解决的问题（“我得到数据了，应该如何处理呢？”“我断开连接了，是否要重连？”）定义一个协议，这样通用组件就能委托给具有更高权限的部分处理。

通过声明这些不同的“问题”，可以建立一种类的使用者能够实现的清晰明确的契约。当他

们要通过实现协议来实现契约时，他们就会知道所有需要准备处理的不同情况。该系统会比涉及返回值、异常等的其他系统灵活得多。甚至比简单的通用回调强大得多，原因就在于 Objective-C 方法声明独有的详尽本质，以及声明一个清晰且显而易见的协议这一操作，如何成为了可复用组件一个自解释的回调 API。如果你用这种方式考虑问题，“协议”这一术语的含义应该比“接口”更丰富，因为“协议”可以被看做一种对过程或行为代码的共识，这与外交和礼仪类似。

好了，理论部分已经够了，现在我们可以试着写写代码了。

7.2 在对象中实现协议

使用协议很简单，只要遵循在处理类时已经见过的很多语法约定就可以了。

根本上说，首先需要创建一个协议声明。你可以在一个现有的接口文件中声明了，在需要为已经存在的对象声明协议，正如之前介绍 `NetworkClient` 时的情况，或者在一个单独的接口文件中声明，因为可能需要在不同情况下使用该协议。协议本身只声明了一个接口并没有提供任何实现。因此，如果为协议声明创建一个单独的.h 文件，就不需要提供任何.m 文件。.h 文件中的接口已经足够了。

在声明协议后，对于任何需要实现该协议的类，必须声明它们要实现该协议。这样编译器就可以确认该类是否实现了程序需要的所有方法。



说明

声明类实现了它所支持的一个协议不是必须的。一些类型的协议实际不需要任何声明，马上就会介绍到。此外，你的类可以选择实现协议的方法而不声明支持该协议。在这些情况下，编译器在编译时会无法确定你的类是否支持该协议，所以必须在运行时做特殊处理，以确保对其调用协议方法的任何对象都实现了这些方法。但是这在其他类表明希望你的类实现该协议时会产生一个编译器警告。

在代码中引用应该实现一个给定协议的对象时，在该对象的类型声明中可以使用一种特殊语法来表明，尽管可能不知道对象实际的类是什么，但你希望它实现给定的协议。如果协议是正式协议，编译时会检查任何存储在变量中的对象，以判断它是否实现了要求实现的该协议的所有方法。如果没有，就会产生警告。

协议可以有必须和可选的方法。在有可选方法的情况下，实现了协议的对象可以不实现其中的可选方法。在这种情况下，在试图调用一个可选方法前需要先确认目标对象是否实现了该方法。如果试图调用一个对象没有实现的可选方法就会发生异常。

7.2.1 声明协议

声明协议要遵循了很多你已经了解的语法标准。从表面上看，它和声明类很相似。通过 `@protocol` 关键字后跟一个要声明的协议名的形式来声明协议。协议默认不从其他协议或类继

承，但是你如果想继承其他协议也是可以的，只需要在所声明的协议名后面的尖括号<>中指定所要继承的协议名。如果这样处理了，实现该协议的类不仅需要实现其所声明的方法，而且也需要实现所继承的任何协议的方法。在@protocol 声明之后，可以声明协议所需的任何方法，这和声明类的方法一致。

在协议定义中，声明协议方法时可以使用两个关键字。第一个是@required 关键字，该关键字表明其后所有的方法是实现该协议的必须方法。这是正式协议的一个默认行为，如果没有指定 @required 关键字，协议中声明的所有方法都默认是必须实现的。

协议声明中的第二个可用关键字是@optional。它表明实现类时可以选择性实现该方法。实现了该协议的类可以选择不实现任何在@optional 关键字后所声明的方法。

在协议声明的末尾，和类一样，可以通过@end 关键字来结束协议声明。用于之前讨论的 NetworkClient 类的一个协议声明的示例如代码清单 7-1 所示。

代码清单 7-1　协议声明示例

```
@protocol NetworkClient
-(void)networkConnector:(NetworkConnector *)inNetConnector
                gotData:(NSData *)inData;
@optional
-(void)networkConnectorDisconnected:(NetworkConnector *)inNetConnector;
@end
```



说明

该协议展示了实际使用的委托模式，这也是第一个参数是发送消息的对象的原因。这在第 17 章中会详细讨论。

此外，如果想要从一个现有的协议中派生一个协议。比如扩展现有协议，你可以通过扩展或者继承现有协议来实现，如代码清单 7-2 所示。

代码清单 7-2　扩展了 IOClient 协议的 NetworkClient

```
@protocol NetworkClient <IOClient>
-(void)networkConnector:(NetworkConnector *)inNetConnector
                gotData:(NSData *)inData;
@optional
-(void)networkConnectorDisconnected:(NetworkConnector *)inNetConnector;
@end
```

协议不能有成员变量。因此，在协议声明中没有声明成员变量的位置。不过不要将此和协议不能访问成员变量的概念混淆了，它当然是可以访问的，但这是和协议实现相关的一个细节，不是协议声明的一部分。在任何类中实现协议的方法时，都可以使用在该类的头文件中所声明的任何成员变量。

7.2.2　声明一个类实现了协议

为了声明一个类实现了特定协议，只需要在类声明中父类后面的尖括号中指定协议名。比如，代码清单 7-3 显示了一个实现了上一节中定义的协议的示例类。

代码清单 7-3　实现 NetworkClient 协议的类

```
@class BusinessLogic : NSObject <NetworkClient>
{
    //成员变量
    NSString *someMemberVariable;
}
-(id)init;
@end
```

类可以同时实现多个协议。这种情况下，可以在尖括号中列举出不同的协议，并通过逗号隔开，如代码清单 7-4 所示。

代码清单 7-4　实现了多个协议的类

```
@class BusinessLogic : NSObject <NetworkClient, DiskClient>
{
    //成员变量
    NSString *someMemberVariable;
}
-(id)init;
@end
```

在本例中，BusinessLogic 同时实现了 NetworkClient 和 DiskClient 协议。

尽管需要引入协议声明中的头文件，但不需要在接口中也声明协议方法。只要声明要实现协议就足以让编译器知道在实现中应该会有什么方法了。



说明

第 8 章介绍的类别也可以声明它们实现一个协议，这和类一样。

7.2.3　声明一个必须实现协议的对象

声明一个用于实现给定协议的实例变量时，通常需要使用 id 数据类型，这样任何对象都可以存储在该变量中。如果需要让编译器确认协议的必选方法在实际存储在该变量中的对象上是否实现了，可以通过数据类型和指定的协议类型告诉编译器。为此，除了 id 数据类型外，还需要在 id 关键字后面的尖括号中指定对象要遵循的协议，如代码清单 7-5 所示。

代码清单 7-5　声明一个实现某个协议的变量

```
id<NetworkClient> *delegate;
```

在本例中，delegate 对象被定义成需要采用 NetworkClient 协议，因此编译器就会希望它实现该协议的必选方法。

任何需要声明变量数据类型的地方都可以使用该语法。这包括方法声明、变量声明以及返回值类型等。



说明

在一些没有要求变量必须实现指定协议的情况下，如果确实需要，可以通过类型转换强制让编译器假定给定对象实现了给定协议。为了将一个给定变量类型转换成给定协议，可以将其类型转换成(id<SomeProtocol>)。如果在变量声明中指定了协议就无需这样处理了，这也是通常推荐的做法。

7.2.4　正式协议和非正式协议

之前我简单提到过，不过在说明如何处理可选方法前需要进一步介绍一下。

实际上有两种类型的协议：正式协议和非正式协议。非正式协议是一种在 Cocoa 和 Objective-C 中仍使用的旧式协议。非正式协议不需要本章前面展示过的正式协议的声明。非正式协议通常声明成 NSObject 类的类别。下一章会介绍类别，因此在这里就不详细介绍了。

由于正式协议可以提供更好的类型安全，并且正式协议有@optional 关键字，支持使用者选择性地将某些方法标成可选，通常来说，如果要在代码中创建一个新的协议应该优先选择正式协议。

通常，最有可能使用非正式协议的场合就是跟旧框架打交道的时候，比如部分 Cocoa 框架。

你会认得这些情形，因为这时文档不会指向一个声明并记录要实现的接口的正式文档，而是你将看到这些协议方法是作为所使用的类的一部分进行记录的。比如，Cocoa 类 NSURLConnection 就在其委托方法中使用了非正式协议。如果查看该类的文档，就可以看到委托方法本身就是在 NSURLConnection 类文档中记述的。它们被标记成 delegate。和 Cocoa Touch 类不同的是，SKPaymentQueue 是为支持 iPhone 上的应用内购买而新增的。它通过 SKPaymentTransactionObserver 协议将其委托方法单独放在一个正式协议中。

如果需要在类中实现一个非正式协议，不需要像实现正式协议那样，在类的声明中表明该类实现了该协议。相反，只要实现想要实现的方法即可，如果它们可用，使用该对象的类就会调用它们。

在使用非正式协议的时候，协议中的所有方法都是可选的，所有在调用之前应该确认目标对象是否实现了这些方法。这点会在下一节介绍。

7.2.5　确定一个对象是否实现了可选方法

在代码中可以通过-conformsToProtocol:对象方法来确认给定类是否实现了特定协议。该方法是在目标对象上调用的，并接收一个协议对象作为参数。为了获取特定协议的协议对象，

你可以使用内置的 Objective-C 指令@protocol()。这与声明协议时使用的@protocol 指令不一样，它在小括号中接收一个参数，该参数就是和想要获取的协议对象对应的协议名。

这样，比如要确定一个给定的对象是否遵循 NetworkClient 协议，你就需要按照代码清单 7-6 进行处理。

代码清单 7-6　运行时确定一个对象是否遵循 NetworkClient 协议

```
-(void)receivedData:(NSData *)inData;
{
    if([delegate conformsToProtocol:@protocol(NetworkClient)])
        [delegate networkConnector:self gotData:inData];
    //否则执行其他
}
```

通常，没有在目标变量类型中指定协议类型时需要这样做。如果在变量的数据类型中指定了协议，编译器会标记没有实现所需协议的变量。



说明

代码清单 7-6 所示的-conformsToProtocol:方法只适用于正式协议。如果你使用非正式协议，请使用代码清单 7-7 所示的 NSObject 的-respondsToSelector:方法

即使你确信给定对象实现了给定协议，但该对象还是可能没有实现协议中的任何可选方法。

记住，如果对象没有实现可选方法但你在该对象上调用了该方法，应用就会崩溃。因此，需要在调用之前判断对象是否真正实现了可选方法。

幸好，所有对象的父类 NSObject 有一个实现了该功能的方法。这个方法就是-respondsToSelector:，它接收一个选择器对象作为参数。

和协议对象一样，可以使用一个特殊的指令将一个方法签名转换成一个选择器对象。该指令就是@selector()指令。和-respondsToSelector:方法一起使用的示例如代码清单 7-7 所示。

代码清单 7-7　检查一个对象是否实现了可选方法

```
-(void)disconnected;
{
    if([delegate respondsToSelector:@selector(networkConnectorDisconnected:)])
     [delegate networkConnectorDisconnected:self];

    //否则实现一些默认行为
}
```

在本例中，首先通过调用-respondsToSelector:方法检查实现了协议的对象是否真正实现了可选方法。如果实现了就调用该方法。如果没有实现，可以选择不执行任何操作或者实现默认行为。

7.2.6　避免协议循环依赖

协议可以在它们自己的声明中引用另一个协议。比如，假想一个协议需要另一个协议作为其中一个方法的参数，如代码清单 7-8 所示。

代码清单 7-8　需要另一个协议的协议

```
@protocol Foo
-(void)someMethodRequiringBar:(id<Bar>)inBar;
@end
```

如果所需的协议（Bar）也需要原有的协议（Foo），如代码清单 7-9 所示，这就会导致两个协议之间的循环依赖。这会导致一个编译器错误。

代码清单 7-9　需要 Foo 协议的 Bar 协议

```
@protocol Bar
-(void)someMethodRequiringFoo:(id<Foo>)inFoo;
@end
```

为了解决该问题，可以给出所需协议的前向声明，这样就不需要包含所需协议的头文件。比如，为了防止循环依赖，可以在 Bar.h 接口文件中加入`@protocol Foo`指令，而不是导入 Foo.h 文件。如代码清单 7-10 所示。

代码清单 7-10　修正后的 Bar 协议

```
@protocol Foo;
@protocol Bar
-(void)someMethodRequiringFoo:(id<Foo>)inFoo;
@end
```

这种情况很少发生，但是知道有这样的一个工具可以在需要时使用很重要。通过在 Bar 协议声明之前加上`@protocol Foo`指令，你可以告诉编译器："相信我，Foo 是一个协议，我会在编译的时候包括它，在这里我就不导入 Foo.h 了"。

7.3　协议使用示例

使用 Cocoa 和 Cocoa Touch 等应用框架时会经常使用协议。最大的使用领域就是委托和数据源对象。通常，在某些情况下，委托对象需要的信息在运行时无法以通用的方式确定，比如需要确定显示给定表格视图中的哪些列。此外，还适用于在后台运行某一进程的情形。你可能想要调用`-start`等方法，它可以马上返回，然后收到一个委托协议的回调方法来通知你该进程已经结束。

实际应用中的例子如代码清单 7-11 所示。在本代码中，应用创建了一个 NSURLConnection 实例，并启动它。然后，当 URL 请求的数据回来时，它就从`-connection:didReceiveData:`方法收到了数据的通知。

代码清单 7-11 一个使用 NSURLConnection 并实现 NSURLConnection 委托协议方法的类

```
@implementation NetworkConnector
-(id)init
{
   if(self = [super init])
    {
     NSURL *url = [NSURL URLWithString:@"http://www.google.com"];
     NSURLRequest *req = [NSURLRequest requestWithURL:url];

     connection = [[NSURLConnection alloc] initWithRequest:req

    delegate:self startImmediately:YES];
    }
    return self;
}
//协议方法
(void)connection:(NSURLConnection *)connection
     didReceiveData:(NSData *)inData
{
    [data appendData:inData];
}
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
   //对那些数据进行处理
}
@end
```

7.4 小结

本章介绍了 Objective-C 中强大的分离机制——协议。利用协议，你能够编写复用性更高的代码，从而使得组件同具体实现分离。通过协议，你可以说“我不关注对象的类型，只要你实现了该接口，我就会和你交互”。协议是 Objective-C 中的一种关键技术。协议使得 Objective-C 有别于且优于其他大多数语言。

第 8 章

扩展现有类

8

本章概要

- ❑ 通过类别扩展现有类
- ❑ 通过匿名类别公开私有 API
- ❑ 使用关联引用添加变量到类

无论一个类或框架设计得如何精良，都不可避免地会遇到一些框架设计者没有预测到的情况。一些开发者竟然说不需要在代码中考虑复用性，因为到头来还是无法实现真正的复用性。对此我不敢苟同，我觉得 Objective-C 提供了一些非常棒的工具，可以提高任何现有语言的复用性。

本章将深入钻研这些工具中最强大的几种。有些可能不是 Objective-C 所独有的，但它们都说明了 Objective-C 的动态性可以使其比其他任何编译语言更灵活、更易复用。

本章介绍的技术主要用于扩展现有类的功能。

8.1　使用第三方框架和类

如果接触过任何编程框架，你可能遇到过这样的情况：语言的标准库提供的现有类实现了所需要的 90%的功能，但却没有提供你真正需要的最后 10%。比如，你使用的类可能不支持正则表达式搜索。

由于这些框架是第三方提供的，所以就无法访问这些框架的源代码。因此，改变现有框架来新增其余 10%的功能是不可能了。即使在可以访问源码的情况下，随着应用发布一个经过修改的、自定义版本的标准库也是一个极其糟糕的作法。

可能考虑的另一种选择就是继承现有类，为需要改变的类创建一个自定义版本。通过这种方式，你可以在该类的自定义版本中添加任何需要的功能。

从表面上看这是一个好主意，而且确实有很多面向对象开发新手都会采用这样的方式来处理该问题。但实际上这会导致其他问题。具体来说，其他阅读代码的人可能会难以理解你的意图。他们可能会暗自发问为什么要创建这样一个自定义字符串类。如果只是添加几个方法，创建一个自定义子类可能就小题大做了。

创建自定义字符串类的弊大于利。此外，合并不同代码库中的不同子类可能会很复杂并有很

多错误。试想两个代码库中有两个差别很小的 `NSString` 的子类。想象一下你需要使用第一个代码库的部分功能以及第二个代码库的部分功能。除了合并两个 `NSString` 子类的困难外，如果两个类有不同的类名，为了使用合并后新 `NSString` 类，其中的一个代码库必须进行大量修改。

例如，在某些情况下为了扩展现有类而使用子类，有点像顾客只需要不同类型的轮毂盖时，你却让工厂直接交付一辆重新定制的汽车。

我不是说子类在所有情况下都不适用。在某些情况下，子类化是复用性问题的完全正确的解决方案。但是，回顾上一章提到的一个面向对象开发原则"优先使用组合而不是继承"，就可以很容易理解，如果像汽车一样设计，类有可复用和可更换的零件的话，对于要复用该类的任何人都会有更大的灵活性和可定制性。

说了这么多，现在就介绍 Objective-C 的某些特性，由此你不用访问源码也无需利用继承就可以从外部添加新功能。

8.2 使用类别

我要介绍的第一种技术称作类别。利用类别你可以通过在类上声明和实现方法来扩展现有类的功能，这些功能可用于整个应用中任何使用原有类的地方。

表面上听起来很酷，但更酷的是在声明一个类别时不需要访问要扩展类的原有代码。而且，类别不是子类。这就意味着，添加的方法实际上是加到了直接操作的类的实现中。只要简单地声明类别，应用中该类的任何用户都可以在类的实例上访问到那些方法。

尽管这是一种糟糕的做法，但还是可以通过类别重写现有方法。这样即使是第三方库在引用该类时也会调用修改后的而不是最初的方法。

如果你从 Ruby 或者 Smalltalk 等其他动态语言转向 Objective-C，你可能很熟悉 mixin 这一概念。mixin 和类别有很多共同点，甚至可以说类别就是 Objective-C 版本的 mixin。

8.2.1 声明类别

类别的声明和类接口的声明相似。也就是，可以通过`@interface` 外加一个想要修改的类名来声明一个类别。在类名后，不是父类，而是可以在小括号中放置一个类别名字。代码清单 8-1 显示了一个在 `NSMutableString` 类上声明的类别。

代码清单 8-1 添加了将 GUID 插入到字符串这一功能的 `NSMutableString` 的类别示例

```
#import <Foundation/Foundation.h>

@interface NSMutableString (GUID)

-(void)appendGuid;

@end
```

在这部分代码中，`NSMutableString` 中添加了一个可以生成 GUID（全局唯一标识符）的

类别。目前，该类别只是简单地在任何字符串的末尾添加 GUID。

8.2.2　实现类别方法

和协议不同的是，只是声明类别的接口是不够的，因为实际上要将方法的实现加入到要修改的类中。这就意味着除了声明类别的接口外，你还必须添加这些方法的实现。代码清单 8-2 实现了在上节中声明的 NSMutableString 类的方法实现。注意@implementation 一行，和声明类一样，用于给出要创建的实现所针对的类名，本例中是 NSMutableString，之后跟着的是小括号中的类别名。

代码清单 8-2　NSMutableString 的 GUID 类别的实现

```
#import "NSMutableString+GUID.h"


@implementation NSMutableString (GUID)

-(void)appendGuid
{
    CFUUIDRef uuid = CFUUIDCreate(kCFAllocatorDefault);
    NSString *str =
        (NSString *)CFUUIDCreateString(kCFAllocatorDefault, uuid);
    [self appendString:str];
    CFRelease(uuid);
}

@end
```

和定义类实现的方法一样，可以在 implementation 块内定义类别的方法。这些方法可以访问类的所有成员变量，可以通过 self 调用类的其他方法，甚至可以使用 super 关键字调用父类的方法。

唯一的限制就是不能声明一个新的成员变量作为类别的一部分。不过有办法向现有类添加变量，这将在本章的后面部分介绍，但是类别不能做到这些。

8.2.3　在头文件中声明类别

和类一样，类别通常在.h 和.m 文件中声明。在某些情况下，将不同扩展类的相似功能放在一个文件中会很方便。比如，如果需要扩展多个类的类似功能，可以将它们放在一个.h/.m 文件中，就可以从概念上将类似的功能单独放在一起。这样将来如果想修改所有类似方法的功能，尽管属于不同类，但仍可以在单个文件中进行。

8.2.4　使用类别

只要简单地声明和定义类别就可以在任何使用扩展类的地方使用它。但是还需要告诉编译器类别的方法是存在的，以避免在编译时产生警告。

为此，只需在任何使用该类别方法的.m 文件中包含声明类别的.h 文件。

换句话说，为了使用 GUID 类别，就必须在任何使用它的编译单元中包含对应的.h 文件，比如代码清单 8-3 所示的 main.m 文件。

代码清单 8-3 使用 GUID 类别

```
#import <Foundation/Foundation.h>
#import "NSMutableString+GUID.h"

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    NSMutableString *aString = [NSMutableString string];

    [aString appendGuid];

    NSLog(@"The guid: %@", aString);

    [pool drain];
    return 0;
}
```

包含了头文件以后，就可以使用这些方法，就好像在原来类上声明这些方法一样。

8.2.5 通过类别拆分功能

使用类别的另一方便之处就是在类变得过于庞大时可以从中提取功能块。在这些情况下，你可能有一个包含很多代码的类。大的类文件在需要修改时会很快变得笨拙。通过提取一部分类的功能到类别，可以使在大量代码中查找需要修改的方法变得更简单。这样，当在现有类中做只影响部分功能的修改时，可以将和该功能相关的所有方法放在一个类别文件中，这样修改起来更容易。

显然，应该让类尽可能简单。不能以类别为借口添加过多的功能到给定类。但是，类确实有一个变庞大的趋势，需要重构时应知道可使用类别这一工具。

8.2.6 扩展类方法

类别不仅仅适用于对象方法。你还可以使用它们来添加类方法。比如，如果你想在 NSMutableString 中添加一个 GUID 类别的工厂方法，你就可以按代码清单 8-4 所示简单地添加一个工厂方法。

代码清单 8-4 在 NSMutableString 中添加工厂方法

```
#import <Foundation/Foundation.h>

@interface NSMutableString (GUID)

-(void)appendGuid;
+(id)stringWithGuid;
```

```
@end


@implementation NSMutableString (GUID)

-(void)appendGuid
{
    CFUUIDRef uuid = CFUUIDCreate(kCFAllocatorDefault);
    NSString *str = (NSString *)CFUUIDCreateString(kCFAllocatorDefault, uuid);
    [self appendString:str];
    CFRelease(uuid);
}

+(id)stringWithGuid;
{
    NSMutableString *ret = [self string];
    [ret appendGuid];
    return ret;
}

@end
```

之前所有的规则都适用，只不过引用类还是引用对象的差别。在类方法中 `self` 关键字指类对象，在对象方法中 `self` 指实例对象。和之前一样，使用类方法时就像它们是在原始类中声明的一样，如代码清单 8-5 中更新后的 main 文件所示。

代码清单 8-5　更新后的 `main` 函数

```
#import <Foundation/Foundation.h>
#import "NSMutableString+GUID.h"

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    NSMutableString *aString = [NSMutableString stringWithGuid];

    NSLog(@"The guid: %@", aString);

    [pool drain];
    return 0;
}
@end


@implementation NSMutableString (GUID)
-(void)appendGuid
{
    CFUUIDRef uuid = CFUUIDCreate(kCFAllocatorDefault);
    NSString *str =
       (NSString *)CFUUIDCreateString(kCFAllocatorDefault, uuid);
    [self appendString:str];
```

```
    CFRelease(uuid);
}

+(id)stringWithGuid;
{
    NSMutableString *ret = [self string];
    [ret appendGuid];
    return ret;
}

@end
```

8.2.7 分析类别的局限性

类别确实有一些限制。类别不能在扩展类中添加任何成员变量。在类别方法的作用域内可以声明和使用局部变量，而且可以使用全局变量或任何传入的参数，但是不能在类中添加任何成员变量。

类别可以通过 super 关键字调用父类方法。但是，没有一种机制支持类别调用它要重写的方法的原始实现。换句话说，通过类别重写现有对象方法时无法调用原始的现有对象方法。

回忆一下之前在介绍如何创建协议时提到的多重继承的危险性，问题就是如果两个父类都定义了同一个方法的实现，在给定的条件下编译器在确定使用哪个实现时就会遇到问题。这个问题不会影响协议，因为协议仅仅是一个接口的声明而不是实现。但是类别就没有那么幸运了。

和多重继承的情况一样，如果两个类别都定义了一个相同类的相同方法，在运行时实际会调用哪个是不确定的。因此，必须避免这种情况。你甚至可以考虑采用一种独特的方法命名前缀系统来避免和其他类别冲突，例如将你的名字的首字母作为方法的前缀。比如，我可能使用-jdAppendGuid:而不是-appendGuid:。

扩展一个系统框架时，方法的命名要很小心。记住，苹果在不断地改进它们的框架，它们可能会添加与你的方法同名的方法。框架中的其他方法可能会依赖于苹果提供的实现，所以你的方法可能会导致苹果的代码失败。所以可能的情况下，你可以在你的类别方法名前加些前缀来避免这类问题。

8.2.8 通过类别实现协议

第 7 章介绍了协议。在那章中我提到了非正式协议的概念。非正式协议是一种通过在 NSObject 上定义类别实现的协议。这样实现时，给定的协议声明不需要一个相应的实现。换句话说，你可以通过 NSObject 的类别简单地声明协议接口，不过你不需要为该类别的这些方法提供实现。非正式协议的采用者必须提供给定方法的实际实现。

由于其在继承层次上的独特位置，所以要在 NSObject 类上定义这些协议。实现类总是从 NSObject 继承，所有在该类上声明的类别都可以选择性地作为接口的一部分实现。

8.2.9　了解在NSObject上创建类别的风险

遗憾的是，在 NSObject 上声明类别有一些风险。你必须知道在 NSObject 上声明的任何类别方法都会成为接口的一部分，如果实现了就成为了运行时中每个类的实现。在一些情况下，这会影响系统的行为，因为有些类会根据是否存在某些具体的方法而改变行为。因此，如果创建的类别所实现的方法属于这种类型，就会对系统某些部分的行为造成你不希望发生的负面影响。记住，在类上声明一个类别时，该类别会在整个应用甚至基础框架中可用。

在 NSObject 上声明类别的另外一个风险就是 NSObject 没有父类。因此，如果调用 super，编译可能可以通过，但会造成运行时错误。

NSObject 是一个“特殊类”，它提供了某种其他类没有提供的运行时功能。从类对象本身而不是类定义的角度看，结果就是 NSObject 类是可以调用对象方法的。这是 Objective-C 中唯一可以这样做的类。为此，NSObject 对 self 对象进行了一些特殊的处理。因此，如果在 NSObject 上定义的类别中使用 self，可以指向类或是对象。

由于这些风险，通常来说，只能在 NSObject 中以接口形式声明类别，而不能提供实现。类别的实现应该只能由子类完成。尽管可以为 NSObject 类别提供实现，但大部分情况下，NSObject 上的类别仅用于声明非正式协议。

8.3　通过匿名类别扩展类

尽管 Objective-C 没有用类声明语法来声明私有方法的机制，有一种定义私有 API 的方法，这只会向类的使用者公开，而不是使用类别的其他人

使用的工具就是匿名类别。本质上，一个匿名类别就是在给定类上定义的没有名字的类别。也就是在定义类别时，不在类名后面的括号中放置类别名，而是让括号为空。在使用匿名类别时，你仅仅声明接口，而不将实现作为类别的一部分。通常，你将类别的声明放在另一个头文件中，可以让可访问到私有 API 的类的使用者导入。实现是在原有类中完成的。你就只是创建了一种从外部访问该实现的机制。

这使得你可以将某一方法作为匿名类别中私有 API 的一部分进行声明，而不需要作为公共类声明中的公共 API 的一部分声明。在导入匿名类别的接口时，编译器希望在匿名类别接口中声明的方法会在被扩展的类中实现。因此，这也使得你可以拥有一个经过声明和编译时检查的 API，而该 API 是私有的，对于类的使用者不可见，除非它们知道如何包含私有 API 类别的头文件。

代码清单 8-6 显示了一个匿名类别声明的示例

代码清单 8-6　Foo+PrivateMethods.h 中的一个匿名类别声明

```
#import <Foundation/Foundation.h>

@interface Foo ()

-(void)somePrivateMethod;

@end
```

代码清单 8-7 显示了 Foo 类自身的实现。注意接口没有声明私有方法，实现却提供了它的定义。

代码清单 8-7 Foo 类实现

```
@interface Foo : NSObject
{

}

@end

@implementation Foo

-(void)somePrivateMethod
{
    //私有功能. ;)
}

@end
```

这里的实现显而易见。匿名 Foo 类别声明了私有方法，并在实际的 Foo 类中实现。

冒着我的面向对象设计证书（如果有的话）被吊销的危险，我觉得高估了私有方法，我无法真正想到一个需要使用它以实现对类的用户真正隐藏私有方法的场合。考虑我所介绍的使用私有方法的情况，如果一个开发人员有足够的动力，他就会轻易“扒开”你的类并访问任何想访问的私有方法，我认为防止这种访问的任何努力都是徒劳的。

但是，在某些情况下，你可能想将特定方法公开给类的特定用户并且不包含在公开 API 中。比如，如果你是单元测试的拥护者，你希望能够在测试“非公有”方法的同时不必在公共类声明中公开这些方法。非公有方法可能是公开 API 中不包含的方法，或者在类之外不使用。在这种情况下，匿名类别是一种极好的解决方案。

8.4 在现有类中关联变量

类别在扩展类时无法加入新的成员变量。这看起来是类别的一个限制，但是实际上并没有这么糟糕。在你真正需要在要扩展的类中添加成员变量时，可以通过继承等轻松实现。但是，也有一些情况你并不想使用继承，不过又很需要在所扩展的类中加入一些额外的变量。幸好，自 Mac OS X 10.6 和 iOS 3.2 开始，Objective-C 运行时内置的一个底层的功能可以实现这一点。这是运行时本身利用的功能，可以在极端情况下使用，例如在不继承并不改变对象的类声明的情况下将一个变量和现有类关联起来。这种技术称作关联引用。无论是否通过类别都可以使用。我会展示一下如何使用，并展示如何在 NSMutableDictionary 上利用它实现一个缓存的排序键值的类别。

在深入介绍之前，我想解释一下可能会造成迷惑的地方。在使用关联引用时，你不是真正地在类中新增一个成员变量，也没有与之关联的属性。它没有和它关联的存取器函数。核心就是关

联引用仅仅是一个和自定义类的具体实例关联的一个存储器。注意我不是说和类关联的存储器。如果你没有显式地将一个引用和自定义类的给定实例关联，那么该实例就不会拥有该引用。

为了给类的实例添加一个关联引用，你可以直接使用 Objective-C 运行时函数 objc_setAssociatedObject。该函数接收 4 个参数：想关联到数据的对象、获取数据的键值、存储引用的值以及一个关联策略，关联策略定义了如何管理存储值的内存。

创建一个关联后，你可以通过 Objective-C 运行时函数 objc_getAssociatedObject 访问关联中存储的值。该函数接收两个参数，数据关联的对象以及关联数据时指定的键值。

最后，如果不再使用关联对象了，你可以通过再次调用 Objective-C 函数 objc_setAssociatedObject 移除关联，不过这次传入 nil 作为要关联的值。

在所有情况下，和值关联的键必须是唯一的。键的实际数据类型是 void*。通常，如果你要使用一个为该键声明的静态变量。这样你就可以确保和该键关联的指针总是指向该指针的单个实例并且是唯一的。

关联策略可以是表 8-1 所示值当中的一个。

表 8-1　关联策略的可能值

值	功　能
OBJC_ASSOCIATION_ASSIGN	指定值将被简单赋值。没有使用保留和释放
OBJC_ASSOCIATION_RETAIN_NONATOMIC	指定值通过非线程安全的方式赋值并保留
OBJC_ASSOCIATION_COPY_NONATOMIC	指定值通过非线程安全的方式复制
OBJC_ASSOCIATION_RETAIN	指定值通过线程安全的方式赋值并保留
OBJC_ASSOCIATION_COPY	指定值通过线程安全的方式复制

可以看出，这些值和声明对象的属性时指定的属性特性很类似。它使用了和关联引用类似的机制。

作为一个在代码中展示关联引用的工作原理的示例，代码清单 8-8 显示了一个在 NSMutableDictionary 上声明的类别，该类别维护一个缓存的、字典键的排序列表。这里出于维护目的定义了几种方法。如果实际实现了该类别，可能会有更好的方式。这里的目的只是简单示范一下用于存储排序键的关联引用的生命周期。

代码清单 8-8　一个排序键的类别

```
@interface NSMutableDictionary (SortedKeys)

-(void)generateSortedKeys;
-(NSArray *)sortedKeys;
-(void)dropSortedKeys;
@end

@implementation NSMutableDictionary (SortedKeys)

-(void)generateSortedKeys;
```

```
{
    NSMutableArray *keys = [NSMutableArray arrayWithArray:[self allKeys]];
    [keys sortUsingSelector:@selector(compare:)];
    objc_setAssociatedObject(self, @"KEYS", keys, OBJC_ASSOCIATION_RETAIN);
}

-(NSArray *)sortedKeys;
{
    return objc_getAssociatedObject(self, @"KEYS");
}

-(void)dropSortedKeys;
{
    objc_setAssociatedObject(self, @"KEYS", nil, OBJC_ASSOCIATION_RETAIN);
}

@end
```

可以看出，创建了排序键数组后，它就存储在 `self` 上的一个关联引用中，而 `self` 即我们操作的字典，如`-generateSortedKeys` 所示。使用完这些排序键后，可以通过`-dropSortedKeys` 方法移除关联引用。

由于没有继承，就务必确保要显示地调用`-dropSortedKeys`（或类似的清理方法），在释放该对象之前释放和其相关的内存。



说明

如果使用更新的 LLVM 1.5 编译器和现代的运行时，它们就包括了在类扩展中声明实例变量的功能，这样你就可以避免大多数的复杂繁琐的操作。为此，就可以作为扩展接口的一部分进行声明，这和在类中声明是一样的。关于启用该行为所需的标志请参照 LLVM 文档。



说明

可以使用 `NSString` 常量（这里我就是使用 `NSString` 常量）作为键，因为如果像我一样在代码中以内联方式定义，它们会被语言定义成一个相互之间的静态引用。

8.5　小结

本章介绍了一些 Objective-C 提供的用小型的可复用组件构建面向对象设计所需的一些独特并强大的工具。如果你是从 C++或者 Java 等动态性低于 Objective-C 的语言转过来的，所展示的方法可能会不太常见，甚至有点神奇。Objective-C 的威力来自这些作为语言自身的一部分并且语言框架完全支持的元编码工具。使用如此富有表现力、如此强大、如此动态的语言会是一种美妙的经历。

8

第 9 章

编 写 宏

9

本章概要

- ❑ 回顾编译过程
- ❑ 使用预处理器 `Define` 创建常量
- ❑ 根据编译器设置编译部分代码
- ❑ 编写编译时控制代码的预处理器宏

本章的主题是宏。宏是 Objective-C 预处理器的一个特殊功能，利用它你可以在编译时执行特殊命令或者替换代码中的特殊值。宏的特殊之处就在于它的命令实际是作为编译过程的一部分执行的。这些命令的结果通常是插入值或者文件等。“宏”这个术语来自于一个小的东西可以展开成更大的东西这一思想，这也正是预处理器宏的功能。

9.1 回顾编译过程

之前介绍过编译过程，但这里我想介绍称作预处理器的这一编译的最初阶段。顾名思义，预处理器是实际开始处理大量源代码之前的一个编译阶段。主要任务就是接收源代码文件，为编译过程做准备。

期间，首先会剔除源代码中的所有注释，并替换成空格。然后进行所有所需的行变换。最后，它会展开所有的预处理器指令，也就是所谓的宏。

预处理器指令就是所有以`#`号打头，后面紧跟着指令本身以及该指令的所有参数的代码行。换句话说，代码清单 9-1 中的所有项都是预处理器指令。

代码清单 9-1　一些预处理器指令或者宏

```
#define FOO 1
#ifdef BAR
#endif
#define BAZ(X, Y) NSLog(@"%s - %s", (X), (Y));
#import <Foundation/Foundation.h>
```

其中的每一项都是预处理器指令。`#define` 定义了一个称作 `FOO` 的常量，其值为 1。`#ifdef` 和`#endif` 定义了一个条件代码块，只有在定义了 `BAR` 的情况下才会编译。`#define BAZ(X, Y)`

NSLog(@"%s - %s", (X), (Y));是一个预处理器函数，它接收参数并记录到日志。最后是#import 指令，之前见过该指令，它加载指定的头文件并以内联方式将其源代码放到源文件中。

这些指令在编译过程中而不是在运行时被展开。所以，它们所影响的是真正编译之前的源代码。换句话说，可以将预处理器宏看做一种编写在编译时控制源代码的程序的方法。

理解宏的工作机制

在编译时控制源代码是一个很有趣的概念。看一下代码清单 9-2。在本段代码中，有一个特殊的字符串常量@"MY_IMPORTANT_DATA"。该常量在访问NSUserDefaults中的项时重复使用。

代码清单 9-2　使用字符串来访问 NSUserDefaults 中的项

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSString *someValue = @"foobar";
   [[NSUserDefaults standardUserDefaults] setObject:someValue
                            forKey:@"MY_IMPORTANT_DATA"];

    //进行一些处理

   NSString *theValue = [[NSUserDefaults standardUserDefaults]
                        stringForKey:@"MY_IMPORTANT_DATA"];



    [pool drain];
    return 0;
}
```

这段代码的最大问题就是在代码中使用了键的常量字符串，这可能产生很多编译器无法捕捉的语法错误。如果输入了错误的字符串，编译器无法辨别出这是一个错误，而是允许其通过并让程序运行。这就会造成很难定位的 bug。

如果你创建的某种编译器可以扩展到字符串的宏，在编译时可以检查语法错误，那么这将非常不错。这是宏的真正作用。代码清单 9-3 显示了利用宏而不是字符串的相同代码。

代码清单 9-3　使用宏

```
#import <Foundation/Foundation.h>

#define THE_KEY @"MY_IMPORTANT_DATA"

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSString *someValue = @"foobar";
```

```
    [[NSUserDefaults standardUserDefaults] setObject:someValue
                                              forKey:THE_KEY];

    //进行一些处理

    NSString *theValue = [[NSUserDefaults standardUserDefaults]
                                 stringForKey:THE_KEY];



    [pool drain];
    return 0;
}
```

可以看到，在新的源码的顶部我们定义了一个名为 THE_KEY 的宏。该宏被定义成 @"MY_IMPORTANT_DATA"。在之后的代码中出现所有的 THE_KEY 在程序编译时都会被替换成 @"MY_IMPORTANT_DATA"字符串。编译过程的这个阶段是透明的，这段代码编译的最终结果会和之前看到的代码清单一样。唯一的区别就是在编写代码时可以利用 Xcode 内置的代码补充，并且编译器会检查到是否正确输入每一个 THE_KEY。比如，如果错将 THE_KEY 输成 THE_KYE，编译器就会检查这种情况并报告一个错误。

代码清单 9-4 显示了程序根据是否定义了给定宏的值（DEBUGGING）而执行不同操作的另一个示例。

代码清单 9-4　基于宏的可选编译

```
#import <Foundation/Foundation.h>

#define DEBUGGING 1

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

#ifdef DEBUGGING
   NSLog(@"Debugging stuff...");
#else
   NSLog(@"Not debugging");
#endif

    [pool drain];
    return 0;
}
```

在本例中，main 函数的代码会检查是否设置了某个值，即 DEBUGGING 宏。如果定义了该宏，也就是如果它有值，就会输出“Debugging stuff...”消息。如果没有定义 DEBUGGING 宏，就会输出“Not debugging.”。这段代码由于使用宏而变得十分强大，可以让部分代码只在调试环境下运行时进行编译。比如，可以用它来控制应用在测试期间连接开发服务器而不是生产服务器。

使用宏的一个很酷的方面就是通过这种方式，可以在编译器设置中使用一些标记，从而严格

按照编译设置来控制这些宏是否被定义。换句话说，你可以将编译设置配置成针对调试环境编译目标时定义该宏，而在编译向用户发布的版本中不定义该宏。同样，所有这些展开都是在编译时进行的。除此之外，记住这些展开都是在源代码中宏所在位置进行的。这很难解释，但通过示例就容易多了，如代码清单 9-5 所示。

代码清单 9-5　一些给出不同期望值的宏示例

```
#import <Foundation/Foundation.h>

#define LOG_LINE NSLog(@"%s:%ld", __FILE__, __LINE__);

int main (int argc, const char * argv[])

{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   LOG_LINE
   NSLog(@"%s %s", __DATE__, __TIME__);
   LOG_LINE

    [pool drain];
    return 0;
}
```

在本段代码中，首先定义了 `LOG_LINE` 函数。该函数在代码中展开后就变成 `NSLog-(@"%s:%ld", __FILE__, __LINE__);`。这会导致程序在运行时记录下 `LOG_LINE` 函数所在位置的文件名和行号。内置宏`__FILE__`和`__LINE__`是编译器自身提供的，展开后就是当前文件名和当前行号。运行程序后你就会注意到，行号在两个不同的 `LOG_LINE` 调用之间是不同的。这只有在宏具备内联展开功能的情况下才可能。另一个这样的示例就是查看 `LOG_LINE` 调用之间的一行代码的输出。内置宏`__DATE__`和`__TIME__`会展开成编译程序时预处理器运行的日期和时间。换句话说，也就是编译程序时的日期和时间。如果一次编译后多次运行，你会发现显示的日期和时间在接下来的运行中不会改变。这是因为源代码中被展开的这个日期和时间实际上是通过代码中这些宏的展开来硬编码实现的。

代码清单 9-6 显示了预处理器展开后的代码。

代码清单 9-6　宏展开后的程序

```
#import <Foundation/Foundation.h>



int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSLog(@"Code/Macros/Macros.m:9");
   NSLog(@"%s %s", "May 18 2010",  "16:00:03");
   NSLog(@"Code/Macros/Macros.m:11");
```

```
    [pool drain];
    return 0;
}
```

表面上说这正是预编译所做的。它接收宏并在代码中将它们展开成任何所定义的内容。

你会注意到这些特殊的宏在宏名前后都使用了双下划线。这是专门为编译器提供的宏保留的，但在代码中不能这样定义。你可以使用这些宏——但不要在自定义的宏名中使用双下划线。

9.2　定义宏

宏定义以#符号打头，后面跟着预处理器指令和任何该指令所需要的参数，如宏名称等。表 9-1 列出了最常用的预处理器指令。

表 9-1　常用的预处理器指令

指　　令	功　　能
#define	用于定义常量和函数等新宏
#ifdef	表示可选编译代码块的开始。如果该预处理器指令的参数被定义成任何值（甚至 0），#ifdef 之后直至#endif、#else 或 #elif 的代码就会被编译并被包括到应用中。如果没有定义该参数并且提供#else 或#elsif 的代码块，就会检查#else 或者#elsif，适当的情况下代码块就会被编译并包括到应用中
#undef	移除之前定义的宏
#import	在该文件中读取并包括另一个源文件。自动防止多次包含该文件
#include	在该文件中读取并包括另一个源文件。不会防止多次包含该文件
#pragma	用于配置编译器和 IDE 注释的特殊宏
#warning	产生一个编译器警告。用于向开发者标志问题
#error	产生一个编译器错误
#if	和#ifdef 类似，表示可选编译代码块的开始，但依赖一个表达式（比如 X > 10），只有该表达式为真时才认为是真的
#else	在#if 或#ifdef 之后使用，提供一个在 if 表达式为假时编译的条件代码块
#elif	在#if 或#ifdef 之后使用，提供一个由条件控制语句确定是否编译的条件代码块
#endif	终止#if、#ifdef、#else 或#elif 代码块

出于本书的考虑，我主要会介绍#define、#ifdef 和其他一些比较常用的预处理器指令。#pragma、#warning、#include 和#error 等指令可以通过察看 GCC 文档更好地了解。



说明

#pragma 指令的常见用法之一就是在代码中添加 IDE 指令，以供 IDE 在标签中使用。苹果在其模板中大量使用该指令。在很多模板中，你可以看到#pragma mark Something 指令。这会使得 IDE 在方法名下拉列表中显示 Something。此外，#pragma mark -这一特殊指令会在列表中加入一条水平分割线。

9.2.1 定义常量

本章展示的第一种类型的宏用于定义一个可以在应用中多个地方重用的常量。的确，这可能是宏的最常见的用途之一。如上个示例所示，我自己就使用这个类型的宏定义用于访问 NSUserDefaults 的键。

要定义一个常量，可以使用预处理器指令#define，后跟要定义值的常量名。在常量名后有一个空格，然后提供一个值，你希望预处理器在源码中将宏展开成该值。预处理器会利用变量名后直至行末的所有文本来展开宏。

在需要定义此类宏并需要跨多行的情况下，你可以通过输入反斜杠后按回车键来实现。这使得编译器在处理宏的时候会将下一行也看作当前行的一部分。

代码清单 9-7 显示了一个通过#define 预处理器指令定义若干个不同的常量的示例。

代码清单 9-7 利用#define 定义常量

```
#define FOO 1
#define BAR @"this is bar"
#define BAZ @"THIS IS A VERY LONG STRING \
AND IT CONTINUES DOWN HERE \
AND HERE."
#define BOZ BAR
```


一个不成文的规定就是宏名都是大写字母。这样就可以在源码中区分宏和普通语句，使得代码更易读。在之前章节中，我提到不能在自定义宏名中使用双下划线。此外，也不能在宏名的开头或结尾使用下划线，因为这些是为编译器保留的。在宏名中间使用下划线是绝对安全的，实际上还有一个不成文规定，宏名中的多个单词就是通过下划线分开的，因为宏名中的空格是不合法的。宏名必须以字母开头，数字是不允许的。在宏名的第一个字母后可以使用数字，但数字就是不能作为宏名的开始。

正如在前一节所介绍的，宏可以在其定义中引用其他宏。在这些情况下，宏首先会被展开成所定义的内容，然后宏定义中的所有宏也会原地展开。比如，上面提到的宏 BOZ 会展开成 BAR，而 BAR 最终会展开成@"this is bar"。

该规则的一个例外就是宏不能是递归的。这意味着，不能在宏定义中使用自身。比如，#define FOO FOO 就不可用。

9.2.2 通过编译传递常量

在本章之前提到过如何在编译设置中定义宏。代码清单 9-8 显示了一个应用示例，该应用需要编译成在调试时连接测试服务器，在非调试时连接到生产服务器。

代码清单 9-8 基于编译设置的条件编译

```
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
```

```
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NetworkConnection *conn = [[NetworkConnection alloc] init];
#ifdef DEBUGGING
   [conn connectToServer:@"http://develop.nowhere.com"];
#else
   [conn connectToServer:@"http://production.nowhere.com"];
#endif

    [pool drain];
    return 0;
}
```

需要注意的是 DEBUGGING 宏实际上不在该源文件中定义。在这个特定示例中，在进行调试编译时通过编译设置来传入该值。如果定义了该值，应用就会连接到开发服务器。如果没有就会连接到生产服务器。

图 9-1 显示了编译设置窗口以及作为编译设置的一部分设置的用于定义值的参数。

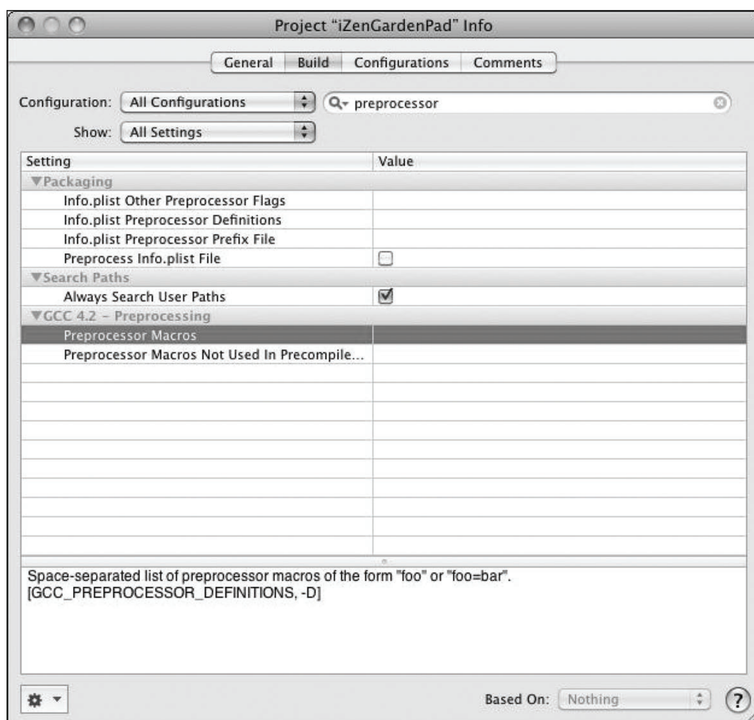


图 9-1 编译设置窗口

用于定义这些预处理器宏的编译设置是 Preprocessor Macros 设置。该设置接收一系列通过=隔开的宏名和值。换句话说，要定义预处理器宏 DEBUGGING，你可以将该编译设置设置成

DEBUGGING=1。这里给定的值是 1，实际上设成任何值都差不多（即使是 0 也是可以的），这样 #ifdef 语句就会返回真。如果使用的是#if 表达式，比如#if DEBUGGING > 10，就会基于编译标志有不同的编译等级。

通过设置调试编译的该项编译设置，在编译源代码时就可以设置 DEBUGGING 宏。如果转到没有该设置的发布编译，就会编译并包含正常的发布代码。

9.2.3　在宏中使用变量

尽管宏的语法功能都是相对比较基础的，但宏可以同函数和方法一样接收参数。这样你就可以创建在宏展开后带有上下文信息的并进行一些创造性工作的复杂的宏。

比如，如果你想创建一个 MAX 宏来返回两个参数之间的较大者，就可能会创建一个如代码清单 9-9 所示的宏。

代码清单 9-9　一个输出变量值的宏

```
#define MAX(X, Y) ((X) > (Y) ? (X) : (Y))
```

在本例中，MAX 宏接收两个参数，X 和 Y。然后它会通过三元操作符比较这两个参数。如果 X 大于 Y 就返回 X，相反如果 Y 大于 X 就返回 Y。

在宏带有参数时，参数在小括号内指定，这和在过程中定义参数类似。不过有些细微的差别。

首先，不需要指定参数的数据类型。这些代码不会被编译。相反，宏在使用之处展开时，参数就会直接插入到展开的宏代码中。因此这里不需要变量类型。

其次，参数列表的左括号必须紧跟在宏名之后。请注意我们之前在定义其他宏时，所要定义的宏的值和宏名是通过空格分开的。因此，如果在宏名和参数列表的左括号之间加入空格，预处理器就会误认为参数列表是宏值的开始，而不是宏名的一部分。

过程和宏在参数方面的另外一处细微差别就是宏值体内的值处理。注意上述示例中在宏的定义体内使用了一些额外的小括号。再次指出，这和宏会在代码中展开并且值也会在宏内展开有关。为了说明这点，考虑一下代码 NSLog(@"Max Value: %ld", MAX(x & 20, 10));的展开情况，如果没有多余的小括号，这就可能会展开成 NSLog(@"Max Value: %ld", (x & 20 > 10 ? x & 20 : 10))。在本例中，操作的优先级表明大于操作比按位与操作的优先级高，因此会先进行 20 和 10 的比较操作而不是预期的 x & 20 然后再与 10 比较。通过在宏定义体内加入额外的小括号，就可以确保操作按预期的顺序执行。换句话说，该代码会实际展开成 NSLog(@"Max Value: %ld", ((x & 20) > (10) ? (x & 20) : (10)));。

9.2.4　字符串化

我喜欢在代码中使用的一个宏技巧就是，在应用运行时记录特定变量的值。这样做很有用，因为无需在调试器中停止运行应用，就可以看到应用的状态并可查看某一时刻的特定值。

为此，我会创建一个接收变量作为参数的宏。由于可能在多个地方对不同变量使用该宏，我

需要得到变量名并紧接着输出变量值。为此，我使用了一种称作字符串化的特殊宏功能。

字符串化接收任何传给它的代码并将其转换成一个 C 字符串。比如，如果给定'x + 10'作为参数，就会变成这"'x + 10'"。这使得它很适合处理这类问题。

代码清单 9-10 显示了一个此类宏。

代码清单 9-10　输出变量值的宏

```
#define LOGVAR(var) NSLog(@"%s: %@", #var, var);
```

这里的关键是在变量名之前加#符号。这就会调用字符串化函数。

按代码清单 9-11 所示在应用中使用该宏，你可以看到它会先输出变量名，然后它会输出变量的值。

代码清单 9-11　输出变量值的宏

```
#import <Foundation/Foundation.h>

#define LOGVAR(var) NSLog(@"%s: %@", #var, var);

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSString *someVar = @"This is the value.";

   LOGVAR(someVar);


    [pool drain];
    return 0;
}
```

LOGVAR(someVar)这行会展开成像代码清单 9-12 的样子。

代码清单 9-12　展开的代码

```
#import <Foundation/Foundation.h>



int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSString *someVar = @"This is the value.";
   NSLog(@"%s: %@", "someVar", someVar);

    [pool drain];
    return 0;
}
```



说明

因为是示例，这段代码就特别简单。本例中，你实际需要创建一个类似这样的宏，可能需要确认传入的变量的类型，这样就可以在输出它时使用正确的格式字符串。

9.2.5 使用条件判断

我们可以使用条件预处理器指令`#if`、`#ifdef`、`#ifndef` 以及它们的相关指令`#else`、`#elif`、`#endif` 选择性地编译部分代码。根据某个变量是否被定义、是否没有被定义或者表达式是否为真来选择性地包括或不包括一整段代码。

第一个这种类型的指令就是`#if` 指令。`#if` 指令用于根据表达式的值允许或者防止某部分代码被编译。表达式可以是任何有效的表达式，使用其他宏、常量、来自围绕宏展开的代码中的变量，等等。关键就是为了编译`#if` 指令后的代码块，表达式的值必须是真。可以通过`#end` 指令来终止`#if` 语句。如果需要在表达式为假的情况下也编译其他代码块，就可以在`#if` 代码块中`#end` 指令之前加入一个`#else`。此外，还可以在`#if` 代码块内部加入`#elif` 指令。

`#elif` 指令也会接收一个表达式并和 `if` 指令一样，只有在表达式为真的情况下才会编译它到下一个指令之间的代码。本质上，整个`#if`、`#else`、`#elif`、`#end` 结构和 `if`、`else` 结构很类似，但它影响到的是应用的哪些部分被编译，而不是程序执行过程中的程序流程。`#ifdef` 和`#ifndef` 指令与`#if` 类似，但`#if` 只检查一个值是否被定义，而不是使用一个表达式。对于`#ifdef`，如果该值被定义代码就会被编译，而`#ifndef` 则是在没有定义该值的情况下会编译代码。

9.2.6 使用内置宏

Xcode 所使用的底层编译器 GCC 有很多可用的内置宏。你可能在本章的一些示例中看过其中的一些。如`__FILE__`、`__LINE__`以及其他宏。要获得更多信息，请查看位于 http://gcc.gnu.org/onlinedocs/cpp/上的 GCC 文档。

9.3 小结

本章介绍了 Objective-C 预处理器，它是一种强大的工具，可用于编写在编译时改变代码的代码。通过它可以做各种事情，例如通过常量防止语法错误，输出变量，甚至通过条件控制编译部分代码等。这个工具可能不太常用，但需要时就很便利。

9

第 10 章 错误处理

10

本章概要

- ❑ 学习不同类型的错误以及应该如何处理
- ❑ 使用返回码返回状态
- ❑ 使用异常处理异常错误
- ❑ 学习合理使用 `NSError`

你希望在运行时不会发生错误，但是你知道这是不可能的。你在努力进行防御性编程，确保所使用的变量都有了它们应该有的值。你可以编写单元测试来确保各种可能的条件都被预见到，并且该问题的解决方案已经内置到了应用中。但是你知道不可能预见所有的问题。你知道不管多么努力地避免应用在真实环境中发生危险，只要应用在真实环境中运行，面对有限内存、有限磁盘空间、在最关键的时候用户中断等实际问题时，就会遇到问题。这时你就要面对错误了。

幸好 Objective-C 有很多内置的错误处理的方法供你选择，这样你就可以编写健壮、可扩展、稳定的代码。你编写的应用在错误发生时能够从容应对，而不是崩溃。

本章会介绍 Objective-C 和 Foundation 框架内置的三种主要机制，它们可以帮助编写能从容面对这些危险的代码，并且可以“正确处理”未预见的问题。在开始前，请先看看在常见的应用中会遇到什么样的错误。

10.1 错误分类

运行常见的应用时会发生三种主要类型的错误。

第一种类型的错误是仅仅包括正确或错误情况的错误。没有其他的附加信息用于查看发生了什么，操作要么成功，要么失败。通常，这是最微小的错误，不会严重地中断程序流程。比如，你的程序需要通过一个互斥锁来获得访问某一共享资源的权限，你的程序尝试获取该资源的访问权限，但失败了，因为程序的其他部分已经在访问该资源了。在这种情况下，你想知道访问资源失败了，并且想再次尝试访问资源。这种类型的错误条件是最微小的。你明确地知道在调用发生错误时应该做什么并且知道什么可能会导致这种错误。

返回码很适合这种类型的错误。理想情况下，返回码可以简单到只是一个布尔值。如果调用

成功，返回 YES，失败就返回 NO。在一些缺少更为复杂的错误处理机制（接下来将介绍）的语言中，需要返回无效情况时，错误码是你的唯一选择。在这些语言中，错误码通常有含义。通常这些语言要求将错误码设置成特定的值，以表示特定的错误条件。通常，Objective-C 不要求这样做，因为还有其他的机制可以给你提供错误码无法提供的更具描述力的错误消息。

第二种类型的错误与第一种类型的错误相反。该类型的错误，如果没有处理会导致数据丢失或者应用失败。这些错误显然比第一种更严重，包括应用无法打开继续运行所必需的资源、数据存储的一致性错误等。设想一下这些错误如此重要，如果不处理的话你就宁可让应用崩溃也不要继续运行，以防止在已经造成的破坏的基础上造成更多破坏。不客气地说，这些错误就是异常条件，所以处理它们的最适合的错误处理机制就是异常处理。幸好在 Objective-C 中这类错误很少见。不过，我还是要在本章稍后展示一下如何处理它们，以及从中如何恢复。

第三种错误的严重性介于前面两种之间。该类型的错误严重到需要沿着栈传回上下文信息给函数的调用方，但是还没有严重到无法恢复的程度。

这是 Objective-C 程序中最常见的运行时错误类型。它如此常见，因此苹果专门为这种错误提供了一个标准的错误处理机制。它使用表示成功的返回码和一个专门的 `NSError` 对象来提供发生错误时的上下文信息。`NSError` 的使用需要些技巧，但是在本章结束后就应该可以很专业地处理此类错误了。

了解如何中断程序流程

知道什么时候中断程序流很重要。

前面所提到的三种错误类型，根据错误发生时你采取的中断程序流的方式，需要使用不同的设计模式。在设计一个可能返回错误的 API 时，考虑一下 API 的使用者以及如何处理可能发生的错误，这些错误会对调用 API 的代码设计造成影响。理想情况下，你要将你的 API 设计成让它的开发者可以在提供最少量的基础结构的同时捕获和处理任何可能发生的错误条件。

如果错误很微小、明显并需要很少的外部（开发者一侧）的干预，你可能会考虑使用返回码来表示特定调用失败了。另一个方面，如果已经发生的错误如此严重以至于你必须完全停止应用以防止对系统照成更多损害，这时可能就需要使用异常。你需要认定如果 API 的用户不处理异常，应用就会崩溃，因为这正是异常的作用。从这个角度看异常，也就是将未处理的异常看做崩溃，就可以看清这类错误条件并帮助你想到真正需要异常的地方（提示：很少见）。

最后，对于很多其他的错误条件——在很微小和很严重之间的错误——`NSError` 机制很可能是正确的选择。它很容易将调用失败的事实沿着栈传递给调用方，同时将确定错误有多严重这一任务交由调用方。

10.2　使用错误处理的不同机制

现在我们就进入如何使用这三种不同的错误处理技术的细节。在接下来的章节中，我会介绍 Objective-C 内置的三种不同的错误处理功能、在代码中如何使用以及如何处理他人代码中的错误。

10.2.1 使用返回码

现在我们已经看过方法和过程都具备退出时返回一个值的功能。这可以通过 `return` 关键字实现。作为方法签名的一部分，需要声明返回值的类型，这会确定方法返回值的类型。

使用返回值来表示成功或者失败是编程语言中一种最古老的错误处理机制。Objective-C 是从 C 语言衍生来的，而在 C 语言中，返回 `int` 返回码来表示不同错误的过程是很普遍的。通常返回码会映射到错误消息，这样就可以通过查看返回值来确定实际发生的错误。尽管可以查找错误代码，但错误码本身通常仅仅是一个数字。这很不方便，因为不同的函数需要使用不同值来表示不同类型的错误，查看这些错误时，需要根据所使用的过程在不同的返回码和错误消息的映射表中查找。

因此，其他的错误处理机制出现了，这种使用返回码的方式变得不主流了。不过，对于简单错误使用返回码仍是值得了解和使用的技术。如果可以避免返回码的最大问题，也就是根据返回码查找错误消息的话，这是最好的，而不是坚持使用布尔值，即在成功时返回 YES，错误时返回 NO 的布尔值。这也是 Objective-C 中使用返回码进行错误处理的方式。

当然这个规则也有例外，比如有些方法成功调用时会返回某个值，有时会返回 `nil`，而不是期望值。幸好，Objective-C 的 `nil` 和 NO，在作为 `if` 语句的控制变量时都被解析成 `false`。这样，在使用就可以认为它们是一样的。

代码清单 10-1 显示了用作硬盘文件包装器的类的示例。在该示例中，预期的情况是硬盘上的数据文件存在并且可读。但是如果不能呢？如果文件不存在，文件就无法被打开。在这种情况下，`-openFileAtPath:`对象方法会返回 `nil`。

代码清单 10-1 一个文件包装类的类定义

```
@interface FileWrapper : NSObject
{
    NSDictionary *contents;
}
-(BOOL)openFileAtPath:(NSString *)inPath;
@end;

@implementation FileWrapper
//这里省略了 dealloc 和其他函数


-(BOOL)openFileAtPath:(NSString *)inPath;
{
    contents = [[NSDictionary dictionaryWithContentsOfFile:inPath]
                                                  retain];
    if(!contents)
        return NO;
    return YES;
}

@end
```

-openFileAtPath:方法实际上使用了 NSDictionary 的一个方法，这个方法使用的正是之前提到的错误处理方式。换句话说，正常情况下 NSDictionary 类方法+dictionaryWith-ContentsOfFile:的返回值是一个 NSDictionary 实例。但是，如果文件不存在，或者无法作为属性列表文件加载，该方法就会返回 nil。代码清单 10-1 所示的方法会检查 NSDictionary 方法的返回值是否是 nil。如果是，自身就返回 NO，否则就返回 YES。

代码清单 10-2 显示了一个使用该类的程序的主函数。

代码清单 10-2　使用文件包装类

```
int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    FileWrapper *wrapper = [[FileWrapper alloc] init];
    if([wrapper openFileAtPath:@"..."])
    {
       //在这里对文件进行一些操作
    }
    else
    {
       //告诉用户文件无法打开
    }

    [pool drain];
    return 0;
}
```

可以看出，在 if 语句中调用了-openFileAtPath:。如果调用返回 YES，就是成功并且可以对文件进行处理。否则就告诉用户文件无法打开。

该示例也说明了返回码的一个问题，也就是无法判断文件为什么不能打开。你所能知道的仅仅是文件无法打开。理想情况下，你希望可以告诉用户具体发生了什么，以及为什么文件打不开。可能是文件丢失了，或者用户没有打开文件的权限。在这种情况下，用户是无法知道这些的。

不过，这确实在错误发生时表示错误的一种最简单的方式之一。

10.2.2　使用异常

现在看看错误处理的另一个极端同时也确实很严重的情况，Objective-C 提供了抛出异常和处理异常的优秀工具。

Objective-C 提供了一些内置的指令用于异常处理。表明一个异常已经发生的做法是抛出或者引发异常。本质上，这包括创建一个 NSException 实例并使用内置的 Objective-C 指令@throw。抛出异常以后，它就会顺着调用栈上行直到被捕捉。为了捕捉一个异常，可以使用 Objective-C 指令@catch。这样需要特殊处理某种类型的异常或者捕捉所有的异常时，就可以使用@catch 指令捕捉 NSException 的某个子类。代码清单 10-3 显示了文件包装器类的示例，不过这次如果文件打不开，-openFileAtPath:方法就会抛出一个异常。

代码清单 10-3　使用了异常的-openFileAtPath:方法

```
-(void)openFileAtPath:(NSString *)inPath;
{
    contents = [[NSDictionary dictionaryWithContentsOfFile:inPath]
                                          retain];
    if(!contents)
    {
        if(![self fileExistsAtPath:inPath])
        {
            NSException *ex =
            [NSException exceptionWithName:@"Error opening file"
                                    reason:@"File doesn't exist."
                                  userInfo:nil];
        }
        else if(![self hasPermissionForFileAtPath:inPath])
        {
            NSException *ex =
            [NSException exceptionWithName:@"Error opening file"
                                    reason:@"Permission error."
                                  userInfo:nil];
        }
        else
        {
            NSException *ex =
            [NSException exceptionWithName:@"Error opening file"
                                    reason:@"Unknown error."
                                  userInfo:nil];
        }
        @throw ex;
    }
}
```

在-openFileAtPath:方法的这个版本中，在确定文件无法打开后，会检查几种可能会失败的典型原因，并为每种情形专门创建了一个异常。准备好了异常以后，就可以通过@throw 指令抛出异常。

在本例中，我使用了默认的 NSException 类来抛出异常。如果想更复杂些，就可以重写这个方法，为每个不同类型的异常情况使用自定义的异常类。

示例如代码清单 10-4 所示。

代码清单 10-4　为不同类型的错误抛出自定义异常

```
-(void)openFileAtPath:(NSString *)inPath;
{
    contents =
        [[NSDictionary dictionaryWithContentsOfFile:inPath] retain];
    if(!contents)
    {
        NSException *ex;
        if(![self fileExistsAtPath:inPath])
        {
```

```
            ex = [FileMissingException
                          exceptionWithName:@"Error opening file"
                                reason:@"File doesn't exist."
                               userInfo:nil];
        }
        else if(![self hasPermissionForFileAtPath:inPath])
        {
            ex = [FilePermissionException
                          exceptionWithName:@"Error opening file"
                                reason:@"Permission error."
                               userInfo:nil];
        }
        else
        {
            ex = [NSException exceptionWithName:@"Error opening file"
                                  reason:@"Unknown error."
                                 userInfo:nil];
        }
        @throw ex;
    }
}
```

注意该方法移除了所有的返回值。这是一个要么成功返回要么不返回任何值的代码示例。换句话说，如果文件可以成功打开，该方法就成功返回，一切也都 OK。如果在打开文件时发生错误，就会抛出一个需要调用者捕捉的异常。



警告

这点再强调也不为过，所以还是再强调一下。在该段代码中，如果抛出了异常但方法的调用栈上没有人捕捉，应用就会崩溃。这点很明确，只有在知道对于应用来说该错误很严重的情况下才能使用异常。

由于捕捉异常很重要，因此应该看看如何处理。代码清单 10-5 显示了更新后的带有合理的异常处理代码的应用主函数。

代码清单 10-5　在主函数中处理异常

```
int main (int argc, const char * argv[])
{
    int retCode = 0;
    @try
    {
        NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

        FileWrapper *wrapper = [[FileWrapper alloc] init];
        [wrapper openFileAtPath:@"..."];

        //对文件执行一些操作

    }
```

```
    @catch (NSException *e)
    {
        NSString *errorName = [e name];
        NSString *errorMsg = [e reason];
        NSLog(@"An error occurred: %@ - %@" , errorName, errorMsg);
        retCode = -255;
    }
    @finally
    {
        [wrapper release];
        [pool drain];
    }
    return retCode;
}
```

异常的工作方式使得它们可以在应用的任何地方中断程序流，可以顺着栈向上直到被捕获。

因此，如果在你所调用的方法中有可能发生异常，你需要将这段代码包装到 try/catch 代码块中。在如上所示代码中，你可以看到该程序做的第一件事就是使用@try 指令。它标志着 try/catch 代码块的开始。使用了@try 指令以后，在紧接着的代码块（通过{}分隔）中的代码就会正常执行到代码块末尾或者直到抛出异常。

如果抛出一个异常，程序流就会马上中断，并跳转到一个通过@catch 指令指定的异常处理程序。

@catch 指令能够捕捉特定类型的异常。如果异常发生，代码就会跳转到@catch 指令位置并查找和所抛出的异常最接近的匹配。在 catch 代码块内重新开始执行代码。

在本例中，catch 代码块只是简单地输出错误消息并将应用的返回码设置成错误状态。如代码清单 10-5 所示，通过捕捉通用的 NSException 就可以有效地捕捉所有异常。

你甚至可以捕捉 id 数据类型而不是 NSException。这样就可以捕捉到所有抛出的对象。

在之前的例子中，不同的异常会在不同的错误条件下抛出，你可以列出用于处理不同类型的异常的所需的单独 catch 代码块。示例如代码清单 10-6 所示。

代码清单 10-6　捕捉不同类型的异常

```
int main (int argc, const char * argv[])
{
    int retCode = 0;
    @try
    {
        NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

        FileWrapper *wrapper = [[FileWrapper alloc] init];
        [wrapper openFileAtPath:@"..."];

        //对文件执行一些操作

    }
    @catch (FilePermissionException *e)
    {
```

```
        // ...
    }
    @catch (FileMissingException *e)
    {
        // ...
    }
    @catch (NSException *e)
    {
        // ...
    }
    @finally
    {
        [wrapper release];
        [pool drain];
    }
    return retCode;
}
```

通过这种方式捕捉异常，必须按照最具体到最通用的方式列出异常，因为运行时环境会执行第一个匹配的 catch 代码块。

由于异常可以在任何时候中断程序，异常发生时要合理地清理所分配的内存和资源就会很困难。幸好，除了 try/catch 结构外，还有一个内置的 Objective-C 的异常处理功能，也就是@finally 指令。

@finally 代码块的工作原理和@catch 代码块一样，只不过是无论是否发生异常都会被执行到。这使得它成为了一个清理内存或者释放其他在@try 代码块中分配的资源的理想位置。无论 try 或者 catch 代码块内发生了什么，@finally 代码块都会被执行。

这种类型的错误处理的优势就是可以将代码分组并使其运行直至一些糟糕的事情发生，然后处理错误情况。此外，错误信息可以非常的详尽。你可以在 NSException 对象上加入各种东西，想要多详尽都可以。

如果你的处理需求比较复杂，需要使用复杂的错误条件，那么可以将 try/catch/finally 代码块进行嵌套。这样一来，在必要时可以使用多重异常处理。



说明

和其他大量使用异常的语言相比，Objective-C 使用异常来进行错误处理的情况要少得多。如果你是从 Java 等其他语言转到 Objective-C 就会不自觉地使用很多异常，我建议你要三思，并看看接下来要介绍的错误处理机制 NSError。



说明

为了让调试器在异常抛出时中断，可以在 Objective-C 运行时在方法 objc_exception_throw 中设置一个断点。这样一来，如果抛出异常，就会触发断点并停止应用。但是你必须小心，因为有些方法会抛出和捕捉异常，而不会让异常顺着栈上行。这是很正常的，不会造成任何问题。

10

10.2.3　使用NSError

在设计 Foundation 框架时，苹果认识到了它们需要一个错误处理机制，它既保留了简单的返回码的简单性，又提供一个用于指出发生了何种错误的更多相关信息的机制。因此就引入了一种称作 NSError 的新的错误处理系统。代码清单 10-7 显示了使用了该技术的文件包装器类。

代码清单 10-7　使用 NSError 的文件包装器类

```
-(BOOL)openFileAtPath:(NSString *)inPath withError:(NSError **)outError;
{
    contents = [[NSDictionary dictionaryWithContentsOfFile:inPath] retain];
    if(!contents)
    {
        if(![self fileExistsAtPath:inPath])
        {
            NSDictionary *errorInfo =
            [NSDictionary dictionaryWithObject:@"File doesn't exist."
                                        forKey:NSLocalizedDescriptionKey];

            *outError = [NSError errorWithDomain:@"FileWrapper"
                                            code:404
                                        userInfo:errorInfo];
        }
        else if(![self hasPermissionForFileAtPath:inPath])
        {
            NSDictionary *errorInfo =
            [NSDictionary dictionaryWithObject:@"Permission Error."
                                        forKey:NSLocalizedDescriptionKey];

            *outError = [NSError errorWithDomain:@"FileWrapper"
                                            code:500
                                        userInfo:errorInfo];
        }
        else
        {
            NSDictionary *errorInfo =
            [NSDictionary dictionaryWithObject:@"Unknown error."
                                        forKey:NSLocalizedDescriptionKey];

            *outError = [NSError errorWithDomain:@"FileWrapper"
                                            code:501
                                        userInfo:errorInfo];
        }
        return NO;
    }
    return YES;
}
```

NSError 是一个在 Cocoa 和 Cocoa Touch 框架中实现的格式化的设计模式。

使用时，需要将方法签名扩展成接收一个 NSError 对象的间接引用。该对象由调用者提供。间接引用实际上就是指针的指针。换句话说，就是指向另外一个指针的指针。在本例中，这是一

个指向在调用者栈中分配的一个变量的指针。在对该变量赋值时，你需要解除间接引用并赋值到实际所指的变量。



说明

有些程序员将间接引用称作“指针的指针”，“传引用”，或者“引用”。我倾向使用“间接引用”，这样就可以同使用 Objective-C++时会遇到的普通的引用区分开。“指针的指针”可能是最准确的描述，但我觉得程序员新手看到“指针的指针”会很迷惑。

间接引用这一概念听起来有点迷惑。这很正常。幸好，只要理解了创建和返回 NSError 对象时所需的语法，底层细节在大多数情况下都不重要。

在方法的参数列表中声明 NSError 的间接引用，可以使用"(NSError **)"语法。两个*表示这是一个间接引用。

1. 创建一个 NSError 对象

在方法中发生错误以后，在返回 NO 之前，你必须首先创建一个 NSError 对象并将其赋给解引用后的 NSError 间接引用。为此，和平常一样创建一个 NSError 对象，在将其赋值给传入的变量时，你需要通过解除引用操作符(*)对变量解除引用。换句话说，就是创建一个新的 NSError 对象并将其赋给传入的 NSError 间接引用，如代码清单 10-8 所示。

10

代码清单 10-8　将 NSError 对象赋给传入的 NSError 间接引用

```
*outError = [NSError errorWithDomain:@"FileWrapper"
                                code:404
                            userInfo:errorInfo];
```

NSError 工厂方法接收 3 个参数。第一个是错误域。这是一个用于表示发生错误的子系统的字符串值。Cocoa 自身提供了几种错误域，比如 NSCocoaErrorDomain、NSPOSIXErrorDomain 等。这些在 NSError.h 头文件中声明。创建 NSError 的一个实例时你可以（也应该）指定自己的错误域。如果需要，错误域字符串就需要使用反向 DNS 表示法，比如 com.yourcompanyname.productname.classname。

第二个参数是错误码参数。该错误码完全是应用特有的，提供了一种在错误对象中指定传统错误码的方式。选择如何使用错误码由你决定，但它必须是一个无符号整数。

2. 了解 NSError 的 userInfo 字典

大多数情况下，错误码和错误域是遗留的参数，现在基本上不用了。NSError 对象的核心是 userInfo 字典。在为此参数创建一个字典时，针对不同的错误信息可以使用几个键值。这些键值如表 10-1 所示。

不是所有的这些键总会存在，此外，NSError 类的用户可以加入与具体域相关的特定键值到字典或者想对与发生的错误相关的数据进行编码。

NSError 可以将显示错误消息给用户所需的全部信息编码，包括描述、错误原因、如何恢

复的建议，甚至是对话框的按钮。这样，我们就可以使用 NSAlert 类方法+alertWithError:来显示一个合适的提醒对话框，并根据 NSError 对象中的数据来显示合适的按钮和文本框。

表 10-1　**UserInfo** 字典的键值

键　值	功　能
NSLocalizedDescriptionKey	本地化后的错误条件的描述，比如“由于不存在，文件无法打开”。也可以通过 NSError 对象方法-localizedDescription 获取
NSLocalizedFailureReasonKey	本地化后的错误的原因。比如“文件不存在”也可以通过 NSError 对象方法 method –localizedFailureReason 获取
NSLocalizedRecoverySuggestionErrorKey	用户可能采取的错误处理方法的本地化后的描述。也可以通过 NSError 对象方法-localizedRecoverySuggestion 获取
NSLocalizedRecoveryOptionsErrorKey	使用对话框向用户展示错误时，用于对话框中的按钮标题的字符串列表。第一个字符串用于最右侧的按钮，然后依次向左
NSRecoveryAttempterErrorKey	一个符合 NSErrorRecoveryAttempting 协议的对象，用于尝试从错误中恢复（仅 Mac OS X 可用）
NSUnderlyingErrorKey	另一个表示实际底层错误的 NSError 对象

3. 使用恢复尝试器

恢复尝试器是一个鲜为人知并且很少用到的 NSError 组件。它提供了一个用于自动尝试从已发生的错误中恢复的对象，但仅在 Mac OS X 上可用。

该对象必须遵循 NSErrorRecoveryAttempting 协议，该协议定义了两个方法。一个是专门供使用模态的、不以文档为中心的用户界面的应用调用的-attemptRecoveryFromError:optionIndex:；另一个是供使用以文档为中心的用户界面的应用调用的-attemptRecoveryFromError:optionIndex:delegate:didRecoverSelector:contextInfo:的两个方法。

恢复尝试器和 OS X 的响应链是协同工作的。为了使用恢复尝试器，可以在响应链上的任何对象上使用-presentError:或者-presentError:modalForWindow:delegate:didPresentSelector:contextInfo:。这两个方法都会为应用展现一个错误，前者用于模态应用，后者用于基于文档的应用。在调用它们时，它们会向用户呈现一个提示，显示从 NSError 得到的信息。在用户单击按钮后，适当的恢复尝试方法就会在恢复尝试器上调用，并传入所单击按钮的索引作为 optionIndex 参数。

如果错误是可以恢复的，-attemptRecoveryFromError:optionIndex:方法会返回一个布尔值。-presentError:modalForWindow:delegate:didPresentSelector:contextInfo:会调用委托对象上提供的选择器方法，该方法与代码清单 10-9 类似。

代码清单 10-9 恢复尝试器回调函数

```
- (void)didPresentErrorWithRecovery:(BOOL)didRecover
                        contextInfo:(void *)contextInfo;
```

恢复尝试器极少使用，通常很可能也不会在代码中遇到。不过这是 NSError 的一个有趣的功能。

4. 在方法中处理 NSError

回到文件包装器的示例，为了充分利用这个新的 NSError 的错误代码，你需要将主函数变成如代码清单 10-10 所示的样子。

代码清单 10-10 使用带 NSError 的方法

```
int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    FileWrapper *wrapper = [[FileWrapper alloc] init];

    NSError *error = nil;

    if([wrapper openFileAtPath:@"..." withError:&error])
    {
        //对文件执行一些操作
    }
    else
    {
        //告诉用户文件无法打开
        //这里你会利用 error 对象
        showErrorToUser(error);
    }

    [pool drain];
    return 0;
}
```

本例的一个重要变化就是可以检查-openFileAtPath:withError:方法的返回值。如果是 NO，错误对象应该包含一个已经初始化的 NSError 对象，该对象提供了向用户显示错误所需的任何信息。



警告

NSError 调用的一些代码示例会将 NSError 设置为 nil，然后检查 NSError 是否被初始化以确认错误是否发生。这是绝对错误的。Cocoa 的某些部分即使在成功的时候也会这样处理 NSError 对象。使用 NSError 的正确模式如上例所示。检查方法的返回值，如果是 NO 或者 nil，再检查错误对象以获取更多信息。

10

这种错误处理方法是最好的。简单并有足够的信息，不会强迫用户进行错误处理。它为 API 的用户提供了足够的灵活性来做正确的事情,而不是伪装成比用户知道更多。

10.3　小结

错误处理是高效程序员应具备的一项关键技能,Objective-C 为你提供了优雅且正确地处理错误情况的足够工具。有了这些可用的工具，就充分利用吧！

Part 3 第三部分

使用 Foundation 框架

本部分内容

- 第 11 章　了解框架之间如何配合工作
- 第 12 章　使用字符串
- 第 13 章　使用集合
- 第 14 章　使用 NSValue、NSNumber 和 NSData
- 第 15 章　处理时间和日期

第 11 章

了解框架之间如何配合工作

本章概要

- 框架介绍
- 学习 Foundation 框架如何和其他框架配合
- 学习如何在项目中加入框架

在 Mac OS X、iPhone 和 iPad 上使用 Objective-C 时，操作系统所提供的可复用的库通常是以框架的形式打包的。这些框架将头文件、文档和动态库打成一个包，其中包含使用其中的代码所需的所有信息和数据。

框架具体如何实现和平台相关。框架可能打包成之前提到的动态库（对于 Mac OS X 系统），或者可以是静态库（对于某些 Linux 或者 BSD 的情况）。由于框架包自身以平台为中心的本性，如何构建一个框架的详尽介绍超出了像本书这种以语言为中心的书的范围。但是，了解一些通常和 Objective-C 一起使用的关键框架及其提供的功能还是很重要的。因此，本章会着重介绍一些可用的框架概括，基于此就可以构建自定义 Objective-C 程序了。稍后你就会清楚原因，因为在连 Foundation 框架都不提及的情况下，要写一本严格以语言为中心的 Objective-C 图书几乎是不可能的。所以本书接下来的部分就会介绍这个关键组件的一些细节。

11.1 了解 Foundation 框架

你可能不知道它，但如果之前练习过本书的示例代码，那么你就已经在应用中使用过框架了。到目前为止展示过的每个示例应用都是一个 Foundation 应用，这就意味着它使用了 Foundation 框架。

大多数语言都有一个标准库。比如 C 语言有 C 标准库。C++将标准库扩展到包括一个标准模板库。Java 也有一个标准库，等等。在大多数情况下，标准库规定了所提供的功能，实现则由平台供应商负责。

Objective-C 作为一种语言不提供此类标准库，但随着时间推移，Foudation 框架逐渐发展成为了一个 Objective-C 所拥有的最接近标准库的东西。它提供了字符串、集合、I/O 等很多和其他语言的标准库一样的功能。

虽然最初是 NeXT/苹果为 NeXTstep 和 Mac OS X 开发的，但它已经成为了其他平台为了真正实现可用的 Objective-C 平台而必须遵循的黄金准则。Foundation 本身就是一个巨大的库，这里无法罗列出 Foundation 提供的每个类和方法的。为了让大家了解一下该库提供了什么，表 11-1 中列出了多个比较常用的类。

表 11-1　Foundation 的常用类

类	功　能
NSArchiver/NSUnarchiver	用于序列化和反序列化遵循 NSCoder 协议的对象
NSArray/NSMutableArray	有序的集合
NSAutoreleasePool	实现了用于保留计数和内存管理的自动释放池
NSBundle	为应用包提供功能强大的接口
NSCalendar	提供处理日历的类
NSData	用于存储通用数据的类
NSDate	用于创建和控制日期的类
NSDateFormatter	提供本地化和格式化日期的类
NSDictionary/NSMutableDictionary	相关的集合类
NSLock	线程锁定
NSError	存储错误信息的类
NSException	异常的基类
NSEnumerator	用于遍历集合的类
NSFileHandle	处理文件 I/O 的包装器类
NSFileManager	封装了创建目录等文件系统相关操作的类
NSGarbageCollector	一个通过面向对象的方式和垃圾回收器交互的类
NSSet/NSMutableSet	无序集的集合
NSNotification/NSNotificationCenter	提供一种方法，用于在运行时发送和接收任何通知
NSObject	所有其他类的基类
NSTask	和操作系统进程交互的类
NSThread	创建并与线程交互的类
NSURL	封装了统一资源定位符的类
NSURLConnection	通过指定协议同互联网上的资源进行连接的网络类
NSString/NSMutableString	Objective-C 的字符串类

Foundation 提供的最重要的类可能就是 `NSObject` 了，它是 Objective-C 中其他所有类的基类。实际上，`NSObject` 提供了作为 Objective-C 一部分的大多数功能，如键值编码、反射、动态调度等。没有 Foundation，Objective-C 就是一个“跛脚”的语言。这两者是相辅相成的。

Foundation 库极其详尽。大多情况下，在考虑自己重新设计一些底层类时首先应该检查 Foundation 是否已经提供了该类。很有可能已经提供了。

至今我还能在 Foundation 和其他框架中发现一些新东西，并惊讶于苹果预见到了所有不常用的边界情况。关于 Foundation 以及任何苹果框架的完整列表，可以访问 http://developer.apple.com 上的苹果文档。

认识其他框架

Mac OS X 中不仅有 Foundation 框架。苹果提供了很多作为标准平台一部分的其他框架以及几百个可以从第三方下载的框架。苹果提供的框架中包括 Appkit、UIkit 和其他特定于网络服务、图像等的功能。Appkit 提供了在 Mac OS X 上构建 GUI 应用所需的类。UIkit 提供了在 iPhone 和 iPad 上构建 GUI 应用需要的类。考虑到本书的目的，我们只会介绍 Foundation。比起 Mac OS X 上的其他框架，该框架具有最强的跨平台支持能力。它除了在 Mac OS X、iPhone 和 iPad 上使用的官方的苹果版本外，在 Linux、BSD 甚至 Windows 上都有第三方开源的实现版本。其中一种实现支持使用 Mac OS X 上的 Xcode 开发应用，并交叉编译成针对 Windows 的可执行文件。在很多方面，Foundation 框架有比专门为此创建的一些标准库具备更好的可移植性和强大功能。

但是由于苹果平台目前是最受欢迎的 Objective-C 开发者的平台，我主要介绍该平台上可用的功能。

11.2　在项目中使用框架

虽然这和 Mac OS X 相关，我还是会简要介绍一下如何在项目中使用框架，这样在需要的时候就会知道如何处理。

苹果提供的框架通常安装在 Xcode 的安装目录中，默认的是/Developer。在该目录下你可以找到一个 Platforms 目录。在 Platforms 目录中，你可以看到和所安装 SDK 的不同开发平台对应的目录。比如，如果安装了 iPhone SDK，就可以看到 iPhone OS 相应的目录。如果没有，就只看到和 Mac OS X 相关的目录。

在 SDK 目录中可以看到一个指向 System/Library/Frameworks 的路径。和这个特定 SDK 相关的特定平台的所有框架都在该目录中。

11.2.1　添加框架

在项目中添加一个框架很简单。你只要右击或者按住 option 单击想要加入框架的目标，然后选择 Add ➤ Existing Frameworks 即可。这样就会弹出一个如图 11-1 所示的框架选择对话框。

该对话框支持从安装的框架列表中选择想让当前选择的目标链接到的任何框架。

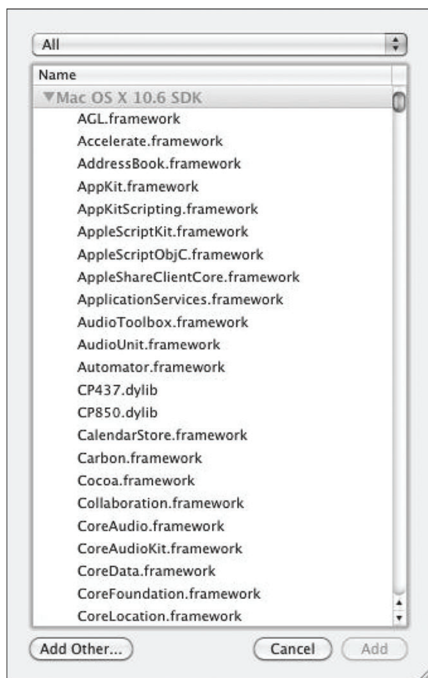


图 11-1　框架选择对话框

11.2.2　包含头文件

记住，简单地链接框架还是不足以让你实际使用其中包含的类和代码。你还需要在源码文件中包含合适的头文件。回忆一下，学习导入头文件时如何通过将导入的头文件名包含在<>中以告诉编译器搜索所链接的框架目录。这样一来编译器就会搜索系统的头文件目录以及任何框架目录。

11.2.3　考虑垃圾回收

在链接第三方框架时要考虑的一个重要因素就是要确保目标框架支持项目所采用的内存管理模型。不是所有的框架都支持垃圾回收和引用计数的内存管理。所以在查看想要链接的框架文档时，确保为所选择的内存管理系统链接正确版本的框架。链接不正确的版本会导致编译错误。

11.3　小结

本章中，我介绍了框架的概念并讨论了 Objective-C 自带的最接近标准库的 Foundation 框架。本书余下的部分将大量使用 Foundation 框架，对框架及其提供的功能有个概念性了解很重要。现在有了这些背景知识，你就会更好地应对接下来的详细编码示例了。

第 12 章

使用字符串

12

本章概要

- 使用 NSString 和 NSMutableString 类
- 理解格式化字符串
- 使用特殊 Objective-C 字符串声明语法

任何一个优秀的标准库都需要一个优秀的字符串类，带有 Foundation 的 Objective-C 也不例外。实际上 Foundation 框架中有一个优秀的字符串类——NSString。和很多 Foudation 的底层核心类一样，有一个不可变版本的 NSString 和一个可变版本的 NSMutableString。这两个类提供了很多处理字符串值的功能。

12.1　了解字符串声明语法

尽管 NSString 和 NSMutableString 有很多类型的初始化函数和工厂方法，由于字符串是 Objective-C 中如此常用的一个类，为了能够简单声明一个字符串，还是显式地创建了一种特殊的语言构造。该构造函数如代码清单 12-1 所示。

代码清单 12-1　Objective-C NSString 的快捷语法

```
NSString *someString = @"this is a string";
```

本质上，编译器一旦遇到@并紧跟着包含在双引号中的字符串就会创建一个静态的包含所提供字符串的 NSString 对象。

任何两个相同字符串值的声明，即使是存储在不同的变量名中，也是指向同一个对象。因此，你可以使用字符串作为键值，比如，在比较一个字符串和该字符串的另一个实例时，就可以用-isEqual:对象方法也可以用比较指针值的==操作符。

关于这点，请看代码清单 12-2。

代码清单 12-2　比较字符串常量是否相等

```
NSString *string1 = @"this is a string";
NSString *string2 = @"this is a string"; //和 string1 相同的对象
```

```
NSString *string3 = [NSString stringWithString:string1]; //创建一个新对象

assert(string1 == string2); //真
assert([string1 isEqual:string2]); //也是真
assert([string1 isEqual:string3]); //真
assert(string1 == string3); //假
```

多数情况下可以通过 Objective-C 字符串构造函数声明字符串，但是 NSString 和 NSMutableString 还是提供了很多初始化函数和工厂方法。常见的一些如表 12-1 所示。

表 12-1 **NSString** 和 **NSMutableString** 的工厂方法

方 法	功 能
+string	构造一个新的空字符串
+stringWithFormat:	使用给定的 printf 风格的格式指定符和任何格式指定符要求的参数构建一个新的字符串
+stringWithCharacters:length:	通过一个新的字符串，其中包含从所提供的 C 风格的数组检索到的长度数组
+stringWithString:	通过给定字符串的值创建一个新的字符串
+stringWithCString:encoding:	通过一个 C 风格的字符串并根据指定编码转换创建一个新的字符串
+stringWithUTF8String:	通过一个 C 风格的字符串并通过 UTF-8 编码创建一个新的字符串。这同调用+stringWithCString:encoding:并将 NSUTF8Encoding 作为 encoding 参数的效果一样
+stringWithContentsOfFile:encoding:error:	使用指定编码创建一个包含指定文件内容的新字符串
+stringWithContentsOfURL:encoding:error:	创建一个包含了 URL 指定的资源内容的新字符串。该资源可以通过给定的协议下载，并通过提供的编码解码。在资源下载过程中该调用是阻塞的
+stringWithContentsOfFile:usedEncoding:error:	创建一个包含了指定文件内容的新字符串。会试图自动判断所用的文件编码类型并利用 usedEncoding 的输出参数通知调用方检测到的编码类型
+stringWithContentsOfURL:usedEncoding:error:	创建一个包含了 URL 所指定的资源内容的新字符串。会下载资源并试图自动判断所用的文件编码并利用 usedEncoding 的输出参数通知调用方检测到的编码类型

其中的每个方法也有一个对应的初始化函数，在选择用这种方式创建字符串时就可以使用。



说明

“工厂方法”指的的可以通过特定参数创建一个对象的类方法。

12.1.1　使用格式化字符串

最经常使用的工厂方法之一就是+stringWithFormat:，该方法使用了 printf 风格的格式化字符串以及参数列表来创建一个字符串。（在本书的很多示例中都会使用到格式化字符串。每次看到 NSLog 调用时就会看到一个格式化字符串实例。）代码清单 12-3 就显示了一个在调用 NSLog 时使用格式化字符串的示例。

代码清单 12-3　NSLog 的格式化字符串

```
NSLog(@"The age of the employee named %@ is %ld", [employee name], [employee age]);
```

使用@""是因为格式化字符串是一个 NSString 的实例。在创建格式化字符串时，你可以使用一个百分号和字符的特殊组合来表明，该参数在传给使用字符串的对象时会被替换成字符串。该格式化字符串包含三部分，第一部分就是格式指定符。格式指定符是一个以%打头的字符的特殊组合，紧跟着有一个或多个数字和字母来指定要在运行时替换成格式字符串的参数格式。你可以使用一个格式指定符来插入对象、整数、浮点数等。格式指定符可以指定参数在置入字符串中的格式。比如，要构建一个格式指定符，用于插入带有两位小数点的浮点值，你可以使用一个%.2f 这样的格式指定符。格式指定符中使用的不同字符很全面，其复杂度和相关技巧可能足以写一本像本书这么厚的书。因此，我不会详细介绍所有的字符。我建议你阅读一下格式字符串的苹果文档。

但是有一个格式指定符我还是要专门介绍一下。那就是%@。该指定符在 Objective-C 中很特别。它和对象参数一起使用。作用就是让格式字符串接收传入的对象作为格式指定符的参数并调用该对象上的-description 方法以获取一个该对象的字符串表示。大多数的 Foundation 类都有一个-description 方法，用于返回一些和目标类相称的值。NSString 的-description 方法就是返回字符串自身。NSArray 的-description 方法返回一个字符串，用来显示数组内容的字符串化表示。NSObject 默认的-description 实现就是返回一个显示该对象指针地址的字符串。如果要创建一个自定义类并想利用类的格式字符串功能，务必要相应地重写-description 方法。

格式化字符串的第二部分称为转义序列。你可以使用转义序列在字符串中插入一些特殊的、非标准的字符串。转义序列的第一个字母总是反斜杠并紧跟一个告诉编译器要插入何种特殊字符的字符。比如，如果要在字符串插入一个换行符号就可以使用转义字符\r。

最经常使用的转义字符有：插入一个新行的\n，换行符\r，双引号\"，制表符号\t。由于转义序列开头都是\字符，因此转义字符\\就表示插入一个\字符。

最后，格式化字符串的第三部分与前两部分不同。这些字符通常都是直接传入到最终结果中。

在解释一个格式化字符串时会遍历格式化字符串，每遇到一个格式指定符就会查找格式字符串是参数本身的函数的下一个参数。然后将该参数的值替换成最终字符串并进行任何格式指定符指明的转换。为了准确说明处理过程，请查看代码清单 12-4。

代码清单 12-4　使用格式化字符串

```
NSString *str;

NSString *cardName = @"Ace";
NSString *cardSuit = @"Spades";

str = [NSString stringWithFormat:@"The winning card is %@ of %@.",
                                   cardName, cardSuit];
 //现在 str 是"The winning card is Ace of Spades. "

   str = [NSString stringWithFormat:@"You have %ld gold!",
                                   [player goldAmount]];
 //现在 str 是"You have 1000 gold! "

   str = [NSString stringWithFormat:@"Your change is: $%.2f.", change];
 //现在 str 是"Your change is $2.43"
```

注意使用格式化字符串时传入的参数使用了一种特别的语法。格式化字符串使得你可以传入一个可变参数列表到+stringWithFormat 方法。参数的个数由所提供的格式指定符的个数所决定。



说明

为了保证 64 位代码的安全，通常在处理整数时使用%ld 格式指定符是一种较好的做法。当整型使用的是 NSInteger 时,会根据芯片架构是 32 位还是 64 位的自动进行修改。使用%ld 作为格式指定符可以确保该格式化字符串可以适应 32 位和 64 位的 NSInteger，而不用考虑其大小。

12

12.1.2　使用其他NSString方法

使用格式化字符串可以通过一种更灵活的方式创建字符串，而不是只能简单地将它们拼接在一起。除了简单构建新的字符串之外，还有格式化字符串不同的应用。比如，可以按代码清单 12-5 所示追加一个格式化字符串。

代码清单 12-5　添加一个格式化字符串

```
NSMutableString *str = [...];

[str appendFormat:@"Your change is: %.2f.", change];
```

通过-componentsSeperatedByString:方法可以将一个字符串拆分成单独的子串。该方法在接收者中搜索给定字符串的实例，并返回一个 NSArray 对象，其中包含由该指定字符串分隔开的子串。该示例如代码清单 12-6 所示。

代码清单 12-6　将字符串拆分成子串

```
NSString *str = @"This is a string of words.";

NSArray *words = [str componentsSeperatedByString:@" "];

//目前 words 是[@"This", @"is", ... ]
```

同样，可以通过 NSArray 的-componentsJoinedByString:方法进行逆向操作。它接收一个字符串数组并将它们合并成其中的每个元素都通过给定字符串隔开的单个字符串。

你可以进行字符查找和替换，使用-rangeOfString:等方法在接收者内查找指定字符串的范围，或者-stringByReplacingOccurrencesOfString:withString:来进行字符串替换。NSString 甚至还有一些有限的正则表达式匹配支持。

总体来说，NSString 是一个非常强大和全面的类，但还有很多功能在这里无法一一介绍。我的建议就是访问 NSString 文档，并查看它提供的所有方法。

12.1.3　使用NSString类别

由于 NSString 已经是一个很详尽的类，苹果选择了将其抽分成不同的可以和 NSString 一起使用的类别文件。类别包括 NSString(AppKitAdditions)和 NSString(UIStringDrawing)类别。严格上说这些不是 Foundation 框架的一部分，它们是 GUI 工具框架 Cocoa 和 Cocoa Touch 的一部分。它们提供了可以用来在窗口和视图上绘制字符串的方法。由于这些不是 Foundation 的一部分，在这里就不介绍了。关于它们的更多内容了请参照苹果文档，或者选择该主题系列书籍中的一本。

12.2　小结

NSString 和 NSMutableString 是 Objective-C 的重要组成部分。关于不同的方法以及如何在代码中使用可以写上好几章。但是，最多能做的就是熟悉 NSString 和 NSMutableString 类的文档。通常，如果需要 NSString 的某些功能，应该都会有。需要时就找找看。

第 13 章 使用集合

13

本章概要

- ❑ 学习如何使用集合
- ❑ 了解可变性和不变性的异同
- ❑ 使用专门的集合
- ❑ 遍历集合的成员
- ❑ 集合的排序和过滤
- ❑ 在集合中使用代码块

Foundation 库提供了大量用于各种用途的类，考虑到在应用中使用到的可能性，很值得深入了解一下几个最基本的类。

这些基础类中有一组称作集合的类。集合是用于管理一组对象的类。每种好的语言都有一个好的集合 API，Objective-C 也不例外。Foundation 框架包含了处理数组、字典、散列表、Set 集合等的类。这些类提供了一种几乎在任何情况下管理和操作一组对象的完整工具集。此外，作为该语言一部分的少量的语法糖使得集合类的遍历、过滤和排序变得简单自然。

13.1 使用数组

我要介绍的第一种集合类就是 `NSArray`。`NSArray` 类用于管理对象的有序集合。一个对象的有序集合是按存储顺序进行维护的一组对象。通常通过遍历或者索引来访问对象的有序集合。

可以通过初始化函数或者众多工厂方法中的一种来创建一个 `NSArray`。创建一个新的 `NSArray` 的示例如代码清单 13-1 所示。

代码清单 13-1 创建一个新的 NSArray

```
NSArray *array = [NSArray arrayWithObjects:@"foo", @"bar", @"baz", nil];
```

`NSArray` 类是不可变的——创建了以后，你就不能改变其内容。但是，由于 Objective-C 没有提供一种确保数组内对象不可变的机制，如果访问数组的一个元素，就可以改变该对象。



说明

集合中的元素不需要是同一类型的。可以任意搭配。

可以通过`-count`方法访问数组中元素的个数，该方法返回一个`NSInteger`。

可以通过快速遍历、索引访问或使用`NSEnumerator`对象顺次访问`NSArray`的元素。这三种方法的示例如代码清单 13-2 所示。

代码清单 13-2　顺次访问数组中的元素

```
NSArray *array = [NSArray arrayWithObjects:@"foo", @"bar", @"baz", nil];

//快速遍历

for(NSString *item in array)
{
    NSLog(@"%@", item);
}

//索引访问
for(NSInteger n = 0; n < [array count]; n++)
{
    NSLog(@"%@", [array objectAtIndex:n]);
}

//使用一个 NSEnumerator 对象

NSEnumerator *enumerator = [array objectEnumerator];
NSString *item = nil;
while((item = [enumerator nextObject]))
{
    NSLog(@"%@", item);
}
```

可以通过`-objectAtIndex:`方法访问数组中的单个元素，该方法返回给定索引处的单个元素。`NSArray`提供了一个便捷方法`-lastObject:`来返回数组的最后一个元素。为了找到一个指定元素的索引值，可以使用`-indexOfObject:`方法，该方法会向数组中的每个元素都发送一条`-isEqual:`消息，并返回第一个结果为真的元素。这两个方法的示例如代码清单 13-3 所示。

代码清单 13-3　访问单个元素

```
NSArray *array = [NSArray arrayWithObjects:@"foo", @"bar", @"baz", nil];


NSString *item = nil;


item = [array objectAtIndex:1]; //现在是'bar'
```

```
item = [array lastObject]; //现在是'baz'

NSLog(@"%ld", [array indexOfObject:@"foo"]); //返回 0
```



警告

NSArray 以及 Objective-C 中的所有集合都是以 0 为基准的。这就意味着第一个元素从索引 0 开始，最后一个元素的索引值比数组长度小一。如果访问一个超出索引范围的元素就会得到一个异常。

除了可以返回单个索引对应的单个元素的方法外，还可以基于一个索引范围返回一组对象。这些方法，如-objectsAtIndexes:和-indexesOfObjects:等方法，和用于访问单个元素的方法的工作原理类似，但接收用于指定期望元素的 NSIndexSet。这些调用可以返回一个和传入的索引想匹配的对象子集。这些方法的示例如代码清单 13-4 所示。

代码清单 13-4 访问一组元素

```
   NSArray *array = [NSArray arrayWithObjects:@"foo", @"bar", @"baz", nil];
NSRange range;
range.location = 1;
range.length = 2;

NSIndexSet *indexSet = [NSIndexSet indexSetWithIndexesInRange:range];

//这会获得[@"bar", @"baz"]的索引
NSArray *subItems = [array objectsAtIndexes:indexSet];
```

NSArray 对象中元素的访问时间，如果没有像这样严格限定，假定最糟糕的情况下是 O(lg *N*)，但大多数操作是 O(1)。线性搜索操作最糟糕情况下的复杂度是 O(*N**lg *N*)。NSArray 类对于存储几乎任何的 Objective-C 对象集合都很有用，包括不属于同一类的对象。但是，NSArray 不能包含 nil 值、构造体和标量。如果需要在 NSArray 中存储一个 nil 值，应该考虑使用 NSNull 的实例。这个类是专门为在容器类中存储“非值”元素而创建的，如果要存储构造体，需要在 NSValue 中包装构造体。存储标量值时也一样，NSNumber 提供了一个可以用来存储标量值的便捷的包装器对象。

13.1.1 使用字典

下一个我要讨论的集合类是 NSDictionary。NSDictionary 类提供了一个用于存储对象的关联集合的容器。关联集合指的是包含相互关联的键和对象的集合。为了访问集合的给定元素，可以通过请求一个和给定键值关联的对象来获取该元素。

通常，字典的键都是 NSString 的实例。这不是必须的。用于键的对象可以是任何类型，只要它们实现了 NSCopying 协议并且是唯一的。字典通过 NSObject 方法-isEqual:同字典的其

他键值比较来确定唯一性，如果试图创建包含两个相同键值的字典，就会抛出异常。

和 NSArray 类似，NSDictionary 可以通过初始化函数或者使用众多的 NSDictionary 类工厂方法中的一个初始化。代码清单 13-5 显示了如何创建 NSDictionary 实例的一些示例。

代码清单 13-5　创建 NSDictionary 实例

```
NSDictionary *dict;

//正常初始化函数

dict = [[NSDictionary alloc] initWithObjects:@"foo", @"bar", @"baz"
                                     forKeys:@"one", @"two", @"three"
                                       count:3];

//工厂方法

dict = [NSDictionary dictionaryWithObjects:@"foo", @"bar", @"baz"
                                   forKeys:@"one", @"two", @"three"
                                     count:3];

//读取 plist 文件并用此创建字典

dict = [NSDictionary dictionaryWithContentsOfFile:@"something.plist"];

//和 NSArray 类似

dict = [NSDictionary dictionaryWithObjectsAndKeys:
        @"foo", @"one",
        @"bar", @"two",
        @"baz", @"three",
        nil];
```

可以通过-objectForKey:这一对象方法访问 NSDictionary 对象的单个元素，该函数会获取到一个和给定键关联的对象。代码清单 13-6 显示了该操作的一个示例。

代码清单 13-6　访问 NSDictionary 对象的单个元素

```
NSLog(@"%@", [dict objectForKey:@"one"]);  //输出'foo'

NSLog(@"%@", [dict objectForKey:@"two"]);  //输出'bar'
```

和使用 NSArray 类一样，该方法有多个变体，可以接收多个键作为参数，从而一次性获取多个对象。其中的一种就是-objectsForKeys:notFoundMarker:，该方法会在给定键没有找到时返回给定对象。该方法的一个使用示例如代码清单 13-7 所示。

代码清单 13-7　使用-objectsForKeys:notFoundMarker:方法

```
NSArray *keys = [NSArray arrayWithObjects:@"one", @"ten", @"two"];

NSArray *items = [dict objectsForKeys:keys notFoundMarker:[NSNull null]];

//items 现在包含[@"foo", NSNull, @"bar"]
```

和 NSArray 一样，NSDictionay 不能将 nil 值、构造体、标量作为对象存储。需要分别通过 NSNull、NSValue 或者 NSNumber 来包装它们。

除了通过键访问一个 NSDictionary 对象的单个元素外，还可以通过-allKeys 或–allObjects 方法来分别获取字典的所有键值或所有对象。这些方法都返回一个 NSArray 的实例，该实例随后会被用在和 NSArray 相同的模式中，用于遍历。也就是说，可以通过一个 for 循环、while 循环等遍历 NSDictionary 实例的所有的键，或者所有对象。示例如代码清单 13-8 所示。

代码清单 13-8　遍历 NSDictionary 对象中的键和对象

```
//两个的输出一样
//遍历对象
NSArray *objects = [dict allObjects];
for(NSString *obj in objects)
{
    NSLog(@"%@", obj);
}

//默认是遍历键
for(NSString *key in dict)
{
    NSLog(@"%@", [dict objectForKey:key]);
}
```

除了使用-allKeys 方法获取一个可以用于遍历的数组外，还可以使用–allObjects 方法。如代码清单 13-8 所示，直接将字典本身作为遍历对象进行遍历，也就是对所有键进行遍历。

最后，可以通过-sortedKeysUsingSelector:方法获取 NSDictionary 对象的一个有序的键数组。由于很快就会介绍集合排序，在这里就不介绍该方法了，而仅仅说明一下它的存在。

和 NSArray 一样，NSDictionary 不能包含 nil 键或者对象，不能包含标量或者结构体。

NSDictionary 在内部通常作为一个散列表来实现。因此，通过键访问一个给定对象通常是 O(1)。此外，这不是显式指定的，你的结果可能会有所不同。

13

13.1.2　使用Set集合

NSArray 类提供了一个可用于有序对象集合的集合，NSSet 提供了一个可以用于无序对象集合的类。通过 NSSet，你可以存储不需要按一定顺序存储的对象。NSSet 是未排序的，因此在访问单独元素时会稍微快点，尽管无法通过索引或者键访问。

再次说明，NSSet 的创建可以通过初始化函数或者 NSSet 众多工厂方法中的一种。代码清单 13-9 显示了如何通过工厂方法创建一个 NSSet 实例并操作 Set 集合的成员。

代码清单 13-9　创建一个 NSSet

```
NSSet *set = [NSSet setWithObjects:@"foo", @"bar", @"baz"];

NSLog(@"%@", [set member:@"foo"]); //输出 foo
```

```
NSLog(@"%@", [set anyObject]); //打印其中一个，但哪个不定

NSLog(@"%@", [set allObjects]); // [@"foo", @"bar", @"baz"]

NSLog(@"%ld", [set containsObject:@"baz"]); //打印'1'
```

存储在 NSSet 中的对象必须响应 NSObject 的-isEqual:和–hash 两个方法。如果存储在 NSSet 中的对象的-hash 方法依赖于对象的内部状态，存储对象在 Set 中时就不能改变。

访问 set 中的对象可以通过方法-allObjects 实现，该方法返回一个包含 set 中所有对象的数组，-anyObject 返回 set 中一个未确定的对象，或者-member:，返回和通过-isEqual:方法确定的输入参数匹配的成员。最后，-anyObject 返回一个未确定的 set 成员。

通过快速遍历或者一个 NSEnumerator 实例可以遍历 NSSet 中的对象。

代码清单 13-10 显示了访问 NSSet 中的对象以及遍历 NSSet 的方法

代码清单 13-10　NSSet 上的操作

```
NSSet *set = [NSSet setWithObjects:@"foo", @"bar", @"baz"];

NSLog(@"%@", [set member:@"foo"]); //输入 foo

NSLog(@"%@", [set anyObject]); //打印其中一个，但哪个未定

NSLog(@"%@", [set allObjects]); // [@"foo", @"bar", @"baz"]

NSLog(@"%ld", [set containsObject:@"baz"]); //打印'1'

for(NSString *item in set)
{
    NSLog(@"%@", item);
}

NSEnumerator *enumerator = [set objectEnumerator];
NSString *item = nil;
while((item = [enumerator nextObject]))
{
    NSLog(@"%@", item);
}
```

Foundation 还提供了一个称为 NSCountedSet 的 NSSet 的子类。通过该便利类，你可以多次加入一个相同的对象。NSCountedSet 可以记录加入给定对象的次数，但是实际上只存储一次该对象。它保存一个加入的给定对象的次数并需要在移除此对象时调用相同次数的-removeObject:。

NSCountedSet 的-count 方法实现返回唯一对象的个数，而不是所有对象加入到 set 的总次数。为了获取给定对象的次数，可以使用-countForObject:方法。

13.1.3　认识可变性

到目前为止我所介绍的每个集合类都是不可变的。在创建集合后，你不能添加或者移除集合对象。如果有这样的限制，集合的用处就不大了。

为此，Foudation 提供了所有这些类的可变版本。可变版本的类名分别是和 NSArray 对应的 NSMutableArray 和 NSDictionary 对应的 NSMutableDictionary 以及和 NSSet 对应的 NSMutableSet。除了不可变版本中用于添加、移除、替换集合中的对象的方法外，这些类还提供了其他方法。

NSMutableArray 提供了-addObject:方法，用于在数组的末尾加入一个对象，-insertObject:atIndex:用于在指定的索引处插入一个对象，-removeLastObject 用于移除数组中的最后一个对象，-removeObjectAtIndex:用于移除给定索引处的对象，-replaceObjectAtIndex:withObject:用于将给定索引处的给定对象替换成另一个对象。代码清单 13-11 显示了在 NSMutableArray 上使用这些方法的示例。

代码清单 13-11　控制 NSMutableArray 的元素

```
NSMutableArray *array = [NSMutableArray array];

[array addObject:@"foo"];
[array addObject:@"baz"];
[array insertObject:@"bar" atIndex:1];
//现在是[@"foo", @"bar", @"baz"]

[array removeLastObject];
//现在是[@"foo", @"bar"]

[array removeObjectAtIndex:0];
//现在是[@"bar"]

[array replaceObjectAtIndex:0 withObject:@"boz"];
//现在是[@"boz"]
```

这些方法还有复数版本，用于对对象数组或者索引范围执行相同的操作。

同样，NSMutableDictionary 也提供了可以操作其对象的方法。比如，加入一个对象到字典可以使用-setObject:forKey:，移除一个对象可以使用-removeObjectForKey:方法。操作 NSMutableDictionary 的示例如代码清单 13-12 所示。



警告

对字典中已经存在的键，调用-setObject:forKey:会利用新的对象重写旧对象。

代码清单 13-12　操作 NSMutableDictionary 中的元素

```
NSMutableDictionary *dict [NSMutableDictionary dictionary];

[dict setObject:@"foo" forKey:@"one"];
[dict setObject:@"bar" forKey:@"two"];
[dict setObject:@"baz" forKey:@"three"];

//dict 目前包含所有三个对象和键
```

```
[dict removeObjectForKey:@"two"];
//dict 目前只有 foo 和 baz
```

最后，NSMutableSet 通过-addObject:和-removeObject:提供了相似的功能。此外，它还提供了添加和移除一组对象的专用方法。比如-unionSet:会将另一个 set 中的所有对象加入到消息接收对象。类似地，-minusSet:会从消息接收方中移除一组对象。

13.2　了解集合和内存管理

在没有垃圾回收的内存环境中使用集合时要很小心。当一个对象从集合中移除时，它就会被释放。这对于开发者有几点含义。

首先由于集合会保留加入它的对象，如果没有合适的理由就不需要在集合外保留这些对象。你可以认为将一个对象加入集合，集合就拥有了它。如果集合被释放，它就会给集合中的每个对象发送 release 消息，因此不需要担心内存泄漏的可能性，因为集合履行了这部分内存管理契约。

第二点，由于对象从集合移除后会被释放，很有可能要继续使用的给定对象已经从集合中移除，并在你不知道的情况下被释放了。代码清单 13-13 显示了一个使用 NSMutableArray 时发生该问题的示例。

代码清单 13-13　从 NSMutableArray 移除一个对象后发生错误

```
NSMutableArray *array = [NSMutableArray arrayWithObjects:@"foo",
                         @"bar", @"baz", nil];

NSString *item = [array objectAtIndex:1];

[array removeObjectAtIndex:1]; //就在这里被释放了

NSLog(@"%@", item); //错误!!
```

该段代码会发生错误是因为尽管已经从数组中获取到对象，但没有保留它。只要没有将该对象从数组中移除，它就会由数组保留并可以随意控制。但是，如果将其从数组移除，它就会立刻被释放。因此，访问该对象的方法就会造成错误。

编写该段代码的正确方式就是在从数组移除它之前保留从数组获取的该对象。代码清单 13-14 显示了修改后的相同代码。

代码清单 13-14　在想继续使用的情况下移除对象的正确方式

```
NSMutableArray *array = [NSMutableArray arrayWithObjects:@"foo",
                         @"bar", @"baz", nil];

NSString *item = [array objectAtIndex:1];

[item retain]; //继续保留!
[array removeObjectAtIndex:1]; //在这里释放 item
```

```
NSLog(@"%@", item); // OK

[item release]; //保留后要记得释放！
```

显然，在垃圾回收的环境中，就不会有这些类型的问题。

使用专用集合

有一些集合类是专门为特定的目标设计的。尽管很少使用，但最好还是在需要的时候知道它们存在。它们是 NSPointerArray、NSHashTable 和 NSMapTable。它们主要用于垃圾回收环境中要求弱关系的专用集合。

这些类提供相似的接口，但没有继承自特定类型的集合。NSPointerArray 对应 NSArray，NSHashTable 对应 NSSet，NSMapTable 对应 NSDictionary。

由于用途的相似以及极少使用，我不会详细介绍每个类。出于示例的需要，看一下其中的一个，比如 NSPointerArray，是很有用的，因为在其构造函数中使用的设计模式和其他两个类足够相似，不需要再进行介绍。

NSPointerArray 可能是专用集合类中最强大的一个。它指定了一个和 NSArray 类似的接口但是支持插入空值和任意指针。此外，在创建一个 NSPointerArray 实例时通过指定某些选项，可以配置数组，从而使其中存储的对象拥有一些具体的内存管理策略。比如，你可以指定对象被空值替换时被垃圾回收器回收。为此，可以通过 NSPointerFunctionsZeroingWeakMemory 选项创建一个弱内存清零的配置。

通过-initWithOptions:或者-initWithPointerFunctions:方法可以在创建 NSPointerArray 实例时指定选项。在使用-initWithOptions:方法时，可以指定所创建的数组遵循由作为参数传递的选项设置的策略。这些选项通过按位或来指定，设定具体策略或者是数组的“个性”。比如，创建一个数组来存储标准 C 风格的字符串，这些字符串需要使用 strcmp 进行比较并且使用 malloc/free 进行内存管理，此时就需要按代码清单 13-15 进行配置。

代码清单 13-15 用于存储 C 字符串的 NSPointerArray

```
NSPointerArray *array = [[NSPointerArray alloc] initWithOptions:
                (NSPointerFunctionsCStringPersonality|
                        NSPointerFunctionsMallocMemory)];
```

在给定实例上你只能设定一个类似的个性选项和一个类似的内存选项。

另外，为了得到最大的灵活性，你可以使用初始化函数-initWithPointerFunctions:来指定一个 NSPointerFunctions 的实例作为参数。该类封装了一个供数组使用的函数，比如求散列值、相等查找、存储和删除。通过该类的实例就可以为每个不同的操作配置一个不同的函数。将这个实例传入到 NSPointerArray 后，它就可以在插入、移除对象时使用你所定义的函数，而不用普通的 retain、release 或者其他在 NSArray 中使用的方法。

真正需要使用 NSPointerArray 的情况可能会很少。但是，它是一个在你需要时可用的工具。

13.3　遍历

集合的元素可以通过快速遍历或者 NSEnumerator 对象进行遍历。要进行快速遍历，只要简单地使用一个标准的 for 循环即可。通过该方式遍历 NSArray 或者 NSSet 的每个元素。使用 NSDictionary 的快速遍历会遍历它的键。

代码清单 13-16 显示了这三种集合类型遍历的例子。

代码清单 13-16　快速遍历集合

```
NSMutableArray *array = [NSMutableArray arrayWithObjects:@"foo",
                         @"bar", @"baz", nil];
for(NSString *item in array)
{
    NSLog(@"%@", item);
}
```

或者，你还可以使用“旧风格”的遍历，即 NSEnumerator。利用该方法，需要获取一个集合的 NSEnumerator 对象，然后反复调用 NSEnumerator 对象方法-nextObject 直至得到一个标志集合末尾的 nil。例子如代码清单 13-17 所示。

代码清单 13-17　使用 NSEnumerator 遍历集合

```
NSEnumerator *enumerator = [array objectEnumerator];
NSString *item = nil;
while((item = [enumerator nextObject]))
{
    NSLog(@"%@", item);
}
```

还可以访问到一个反向遍历器，可以用于反向遍历一个集合的元素。可以在传统的 while 循环以及快速遍历中使用，如代码清单 13-18 所示。

代码清单 13-18　遍历技巧

```
NSEnumerator *enumerator = [array reverseObjectEnumerator];
NSString *item = nil;
while((item = [enumerator nextObject]))
{
    NSLog(@"%@", item);
}
for(item in enumerator)
{
     NSLog(@"%@", item);
}
```

这同样适用于正向遍历器，但由于这是快速遍历的标准行为，因此会很少用到。

在遍历时改变集合的内容是很危险的，会导致遍历变得无效并且导致错误。因此，不能这么做。

13.4　向元素发送消息

另一个常见的需求就是遍历集合中每个元素的同时对其调用一些方法。这可以通过对象方法 -makeObjectsPerformSelector: 和 -makeObjectsPerformSelector:withObject: 实现。这些方法接收一个选择器对象，该对象指定了在集合中的每个对象上要调用的方法。在后一个方法中，通过 object 参数给定的参数也传入了每个方法调用中。

代码清单 13-19 显示了一个使用该方法来遍历游戏引擎中的一个集合并更新它们的位置的示例。在本例中，向每个元素中传入游戏状态就可以使用它。

代码清单 13-19　让集合中的所有对象执行某个动作

```
NSArray *gameObjects = [...];

GameState *gameState = [...];

[gameObjects makeObjectsPerformSelector:@selector(updatePosition:)
                             withObject:gameState];
```

这会让数组中的每个元素都接收到一个带有 gameState 参数的-updatePosition:调用。

13.5　排序和过滤

NSArray 和 NSMutableArray 可以从众多的排序和过滤功能中受益。由于 NSArray 是不可变的，有可以获取排序或者过滤后的数组的一个副本的方法，NSMutableArray 支持就地排序数组。出于本处讨论的目的，我主要介绍 NSMutableArray，但你也应该知道同样可以在 NSArray 上访问任何不可变方法。

数组排序可以通过-sortUsingDescriptors:、-sortUsingFunction:context:或者我个人喜欢的-sortUsingSelector:方法进行，每种方法接收不同类型的排序对象作为参数用于执行排序。

其中的第一种，-sortUsingDescriptors:接收一个 NSSortDescriptor 实例作为参数。为了创建一个参数，你需要指定用于排序对象的属性的键路径，是升序还是降序，最后可以选择一个选择器，它使用进行比较的属性作为参数。如果没有提供选择器，默认使用标准的-compare: selector。

换句话说，如果有一个 Employee 对象数组并想基于入职日期排序。本例中实际需要排序的属性是入职日期属性，你可以使用代码清单 13-20 所示的代码进行排序。

代码清单 13-20　通过-sortUsingDescriptors:排序一个员工数组

```
NSMutableArray *employees = [...];

NSSortDescriptor *descr =
[NSSortDescriptor sortDescriptorWithKey:@"employmentDate"
                              ascending:YES];
```

13

```
NSArray *descriptors = [NSArray arrayWithObject:descr];

[employees sortUsingDescriptors:descriptors];
```

在本例中，我只使用单一的描述符，实际上可以提供多个描述符使用多个标准进行排序。该参数是一个数组，所以你只要将它们放入 `NSArray` 对象并调用方法。

第二个方法是`-sortUsingFunction:context:`。在需要使用函数指针进行比较时使用。作为参数传入的函数指针的格式是 `NSInteger comparisonFunction(id obj1, id obj2, void *)`。前面两个参数是用于比较的两个对象。第三个参数是上下文信息，它是提供给该方法的附加参数，不需做任何修改，按原样给出即可。该技术在需要传入一些额外的外部信息用于比较时很有用。代码清单 13-21 显示了使用该技术进行数组排序的示例。

代码清单 13-21　通过函数进行数组排序

```
NSInteger sortByEmploymentDate(id employee1,
                               id employee2,
                               void *ctx)
{
    return [[employee1 employmentDate]
             compare:[employee2 employmentDate]];
}

NSMutableArray *employees = [...];

[employees sortUsingFunction:sortByEmploymentdate context:nil];
```

第三种技术，使用`-sortUsingSelector:`方法，接收一个选择器对象作为参数。在使用时，数组遍历所有的元素并调用给定的选择器作为比较方法。目标选择器是作为所有数组元素所属类的对象方法实现的。它接收数组的其他一个元素作为参数，并返回一个 `NSComparisonResult` 值，根据方法的接受者和传入的对象是相等、升序还是降序返回 `NSOrderedSame`、`NSOrderedAscending` 或者 `NSOrderedDescending` 这几种值。代码清单 13-22 显示了通过该技术对该员工数组进行排序的代码。在本例中我还展示了 `Employee` 类，以说明如何创建一个要被调用的方法。

代码清单 13-22　通过`-sortUsingSelector:`对员工数组排序

```
@interface Employee
{

}
-(NSComparisonResult)compareEmploymentDate:(Employee *)other;
@end

@implementation Employee

-(NSComparisonResult)compareEmploymentDate:(Employee *)other;
{
```

```
    return [[self employmentDate] compare:[other employmentDate]];
}

@end

[employees sortUsingSelector:@selector(compareEmploymentDate:)];
```

要再次说明的是，其中的每个方法都是就地进行可变数组的排序。NSArray 类通过 -sortedArrayUsingSortDescriptors:、-sortedArrayUsingFunction:context: 和 -sortedArrayUsingSelector:方法分别提供了一个相应的、不可变的方法。



说明

可以使用代码块进行数组排序，下一节会详细介绍。

要过滤一个数组，可以使用-filterUsingPredicate:方法，该方法接收一个 NSPredicate 对象作为参数。NSPredicate 对象支持通过简单的查询语言指定用于过滤数组所要满足的条件。可以通过使用查询字符串指定查询条件。NSPredicate 中使用的查询语言和 SQL 类似，但不是 SQL 语句。利用它可以指定"firstName =='John'"或者"birthDate >= '01/01/2001'"之类的条件。在这些示例中，使用指定操作比较命名属性和给定值，如果返回 true 该对象就被包含在结果集合中。如果返回 false，该对象就被过滤掉。

因此，为了从上例中的所有员工中过滤出在公司工作多于 5 年的员工就可以进行代码清单 13-23 所示的操作。

代码清单 13-23　过滤一个数组

```
NSPredicate *predicate = [NSPredicate
                          predicateWithFormat:@"employedForYears >= 5"];

NSArray *seniorEmployees = [employees filterUsingPredicate:predicate];
```

这里假设你创建了一个对象方法，用于在所有员工中计算员工工作年限并查找该值大于或等于 5 的员工。

13.6　在集合中使用代码块

NSArray 支持利用代码块进行变换。在代码块一节中我简要介绍过这些，但出于完整性的考虑，我会在此再次介绍一下。-enunerateObjectsUsingBlock:方法接收一个代码块作为参数并执行该代码块，遍历时将数组的每一个元素都传入代码块中。该方法的另一种形式就是 -enumerateObjectsWithOptions:usingBlock:，它可以接收一个 options 变量，利用该参数可以指定以何种方式进行遍历。该参数是一个按位或的值，它可能是指定了并发遍历或逆向

遍历的 NSEnumerateConcurrently 或 NSEnumerateReverse 标志中的一个或两个。第 5 章介绍了如何通过执行代码块来遍历数组元素。在代码清单 13-24 中，我会直奔主题，用 -enumerateObjectsUsingBlock:方法来执行相同的操作。

代码清单 13-24　对 NSArray 的元素执行映射操作

```
__block NSMutableArray *result = [NSMutableArray array];

void (^theBlock)(id obj, NSUInteger idx, BOOL *stop) =
^{
    [result addObject:transformObj(obj)];
}

[array enumerateObjectsUsingBlock:theBlock];
```

代码运行后，结果数组应该包含经过了 transformObj 所指定的变换的原始数组中的所有元素。

可以使用-enumerateObjectsAtIndexes:withBlock:方法在一个数组的子集内实现相同功能。可以通过-indexesOfObjectsPassingTest:方法访问一个数组的子集。利用该辅助方法可以将代码块传递给给定对象。如果代码块返回 YES，该对象的索引就会加入到返回数组中，否则就不会。使用代码块就可以轻松实现过滤。

执行相似过程的另一种方法就是使用-makeObjectsPerformSelector:或-makeObjects PerformSelector:withObject:对象方法，这两个方法遍历数组元素并调用每个对象的给定选择器。和接收代码块的方法相比，这个方法的劣势就在于选择器需要在数组中的对象所在类上定义。在有些情况下这就很困难，可能会迫使你为此创建一个类别来扩展一些第三方类。

13.7　小结

集合类是所有标准库的一个重要部分。幸好 Objective-C 有一组优秀的集合类简化了对象组的处理。有了用于有序集合的 NSArray、关联集合的 NSDictionary 以及无序集合的 NSSet，你就有了所有的工具。

第 14 章

使用 NSValue、NSNumber 和 NSData

14

本章概要

- ❑ 集合中使用的数据类型的装箱
- ❑ 使用 NSNumber 和 NSValue
- ❑ 使用 NSData 和 NSMutableData

正如第 13 章强调的，使用集合时，集合只能存储有效的 Objective-C 对象。集合不能存储标量、结构体或者其他任意底层数据。这是一个不方便的地方，但 Foundation 框架的设计者们已经预见并处理了该问题。

为了在集合中存储标量和结构体，需要对这些值使用包装器类。换句话说，支持在对象中存储值的类。Foundation 框架为此提供了 NSValue、NSNumber 和 NSData 这 3 个主要的类。

NSValue 类是这三者当中最简单的类，为几乎所有的 C 数据类型提供了一个可以存储任意值的底层接口。比如，你可以在其中存储构造体，可以存储范围等。数据被存储到 NSValue 实例中以后，你就可以在集合对象中使用该 NSValue 实例了。由于 NSValue 相对比较底层，它不能提供一些更高层抽象所带来的便利。它的设计目标是灵活但是在功能上也有限制，因为它只能存储一些简单的在栈上分配的数据。NSNumber 是 NSValue 的一个子类。相比 NSValue 数据编码系统，它提供了更高层面的抽象，NSValue 数据编码系统专门用于存储数字。它为不同标量类型提供了众多工厂方法，用于更方便地构建和操作 NSNumber 对象。使用 NSNumber 通常比直接使用 NSValue 更容易，应该在可能的情况下优先选用它。

之前提到了 NSValue 只能存储简单的在栈上分配数据。实际上，在 NSValue 中存储动态分配的数据的引用也是可能的。可以用此来追踪集合中动态分配的数据。但是，这样处理时就必须用其他代码来跟踪实际数据本身，这样你就可以对它进行分配和释放了。像这样手动管理内存是不方便的。幸好，还有一个包装任意动态数据的类。这个类就是 NSData。该类为动态分配的字节缓冲区提供了一个面向对象的接口。可以用该接口来存储大块的字节，直接分配或者从一个分配的缓冲区复制字节。你还可以使用它将数据写到硬盘上。

14.1　使用 NSValue 和 NSNumber

现在，我就开始在接下来的几节中逐个介绍这些类并展示如何使用它们。先从 NSValue 和 NSNumber 开始，然后介绍 NSData 和 NSMutableData。

14.1.1　通过NSValue包装任意数据类型

之前提到过 NSValue 是用来存储任意数据类型的。创建 NSValue 时要提供一个想要存储值的指针，以及一个表明其类型的 C 字符串。告诉 NSValue 实例数据类型是很重要的，因为这样就会告诉它，要获取所有数据需要读取多少字节。幸好，Objective-C 提供了一个特殊的指令 @encode()来返回平台上给定数据类型的合适编码。代码清单 14-1 显示了如何使用任意结构体创建一个 NSValue 实例，传入一个结构体实例的地址作为 value 参数的指针，并通过@encode()指令来查找合适的数据类型。

代码清单 14-1　为任意结构体创建 NSValue 实例

```
typedef struct
{
    int someMember;
    float someOtherMember;
} MyDataType;


MyDataType item;
item.someMember = 10;
item.someOtherMember = 500.3;
NSValue *boxedStruct = [NSValue value:&item
                         withObjCType:@encode(MyDataType)];
```

该技术适用于自定义或者框架提供的任意结构体。比如，可以使用代码清单 14-1 中的代码来为 NSRect 和 NSSize 这两个 Foundation 结构体编码。

NSValue 可以用于存储整型、浮点型等，尽管对于这两种类型的值，NSNumber 可能是一个更好的选择。

还可以在 NSValue 中存储动态数据的指针。为此，可以按代码清单 14-2 所示存储指针地址。

代码清单 14-2　在 NSValue 中存储指向动态数据的指针

```
char *foo = malloc(1024);

NSValue *boxedPointer = [NSValue value:&foo
                           withObjCType:@encode(char **)];
```

需要注意的一个要点就是实际要存储的是指针本身而不是数据。因此，需要确保将它存储到 NSValue 中之后动态分配的数据不会被释放。

14.1.2 通过NSNumber包装数字

在处理整型、浮点数等数字时对于一个更高层面的抽象，NSNumber 类提供了一些能够自动进行类型转换和类型判断的额外的工厂方法和存取器函数。使用 NSNumber 简单到只需使用数值调用合适的工厂方法。代码清单 14-3 显示了一些示例。

代码清单 14-3 创建 NSNumber

```
int someNumber = 110;
float someFloat = 500.3;
NSNumber *theNumber = [NSNumber numberWithInt:someNumber];
  NSNumber *theFloat = [NSNumber numberWithFloat:someFloat];
```

14.1.3 通过NSDecimalNumber进行算术运算

尽管为了进行算术运算可以简单地获取 NSNumber 中的底层值，但是有时就想仅通过 NSNumber 对象进行一些简单的操作。为此，Foundation 提供了 NSDecimalNumber 类。

NSDecimalNumber 类是 NSNumber 的子类，它提供了执行简单的十进制算术运算的方法。它有很多方法，如 -decimalNumberByAdding:、-deciamlNumberBySubtracting:、-decimalNumberByRaisingToPower:等方法。这些方法使得使用-makeObjectsPerform-Selector: withObject:等 NSArray 方法来对集合上的所有成员进行算术运算变得更容易。



说明

NSDecimalNumber 是不可变的，所有这里提到的任何算术操作都会返回一个新的 NSDecimalNumber 实例。

如何利用这些方法来为员工数据库中的所有员工发奖金的示例如代码清单 14-4 所示。

代码清单 14-4 给所有的员工发 5000 美元的奖金

```
NSArray *employees = ...;
  [employees
makeObjectsPerformSelector:@selector(addToSalary:)
withObject:[NSDecimalNumber numberWithFloat:5000.0]];
    //addToSalary:的可能实现

    -(void)addToSalary:(NSDecimalNumber *)inRaise
    {
       self.salary = [self.salary decimalNumberByAdding:inRaise];
    }
```

由于 NSDecimalNumber 能够存储很大的值（大到 38 位 x 10^+/–128）。这样进行一些大数值运算时也很方便，但是直接使用 C 的标量值会比通过 NSDecimalNumber 更快，所以需要慎重选择该方法。通常来说，这只在集合中使用。

14.2　使用 NSData 和 NSMutableData

在使用二进制数据块时，Foundation 提供了 NSData 和 NSMutableData 类用于处理数据的面向对象接口。这些类可以用于管理缓冲区的分配和释放，并提供一个在集合类中存储数据的对象包装器类。它们同时也提供了一个将数据写到文件以及通过套接字通信进行数据传输的接口。

14.2.1　创建NSData对象

可以利用之前分配的现有底层数据结构体或从支持 NSCopying 协议的 Objective-C 对象类型复制数据来创建 NSData 对象。

为了利用 C 数据结构体的原始数据创建一个 NSData 对象，可以使用+dataWithBytes: length:工厂方法，该方法接收一个指向数据缓冲区的指针并复制数据缓冲区中的字节到 NSData 对象。如果想直接访问缓冲区中的数据而不用复制，就可以使用+dataWithBytes: NoCopy:length:工厂方法，这样不复制数据就可以创建一个 NSData 对象，实际结果就是会生成一个 NSData 对象，可以直接通过缓冲区访问所提供的原始内存。本例中，由于 NSData 对象会通过 free 函数释放数据，所以所提供的字节必须通过 malloc 创建。由于 NSMutableData 支持对它所包含的数据进行修改，当指向一个外部分配的缓冲区时，修改数据就会有问题。因此，在使用 NSMutableData 时对象会复制数据，而无视是否指定要复制。代码清单 14-5 就显示了一个利用预先分配的字节缓冲区创建 NSData 对象的示例。

代码清单 14-5　创建一个 NSData 对象

```
char *buf = malloc(1024);

NSData *data = [NSData dataWithBytes:buf length:1024];
```

NSData 对象最常用于访问存储在文件中或者网络资源中的数据。可以通过工厂方法+dataWithContentsOfFile:利用文件的内容创建一个 NSData 对象，该方法接收一个文件路径作为参数。对于网络上的资源则可以使用工厂方法+dataWithContentsOfURL:。该方法会通过指定 URL 中提供的协议访问网络并下载资源，之后将资源中可用的原始数据作为 NSData 对象的原始数据。通常，和大多数这类的便捷方法类似，下载过程会阻塞当前线程直至完成，因此要小心使用该方法。NSData 还提供了-writeToFile:atomically:和-writeToURL:atomically:方法将数据写入硬盘。-writeToURL:atomically:仅支持写入本地文件的 URL。这两种方法的第二个参数都是指定写入是否是原子性操作。在要写入的数据相当大时，应用可能在写入数据的过程中终止。这会导致硬盘上的文件受损。atomic 参数指定该文件首先被写入临时文件，并

在临时文件上的文件操作完成后复制到最终位置。通过该标志就知道原始文件仅在替换文件可以完全写入成功的情况下被替换。

14.2.2　访问**NSData**对象中的生数据

NSMutableData 类为包含着想要操作的字节的 NSData 对象提供了一个面向对象的接口。你可以通过-appendBytes:length:和-appendData:方法来增加字节，可以通过-replace-BytesInRange:withBytes: 替换字节，通过 -setLength:. 方法来截断或者扩充 NSMutableData 的缓冲区。如果只是想将缓冲区的某部分置为 0（将字节设置成 0），可以使用 -resetBytesInRange:方法。它提供了任意操作文件或者结构体中的原始数据所需的工具。代码清单 14-6 展示了可以使用该功能来读取硬盘中的文件、修改指定字节，然后将其写回硬盘。在本例中，目标文件是一个使用了硬编码文件格式的遗留的游戏数据保存文件，它会将特定的值存储在数据的特定位置。

代码清单 14-6　修改游戏保存文件的原始字节

```
int goldOffset = 617; //文件中偏移 617 的位置
int goldLength = 4; //用于存储 gold 值的 4 字节空间

  NSRange goldRange = NSMakeRange(goldOffset, goldLength);
  NSMutableData *gameData = [NSMutableData dataWithContentsOfFile:@"..."];

   [gameData replaceBytesInRange:goldRange withBytes:newGoldValue]; [gameData
   writeToFile:@"..." atomically:YES];
```

可以看出，NSData 提供了一个访问原始数据的方便的底层接口。它还提供了一个在集合类内部存储数据的一个很方便的包装器，这与 NSNumber 和 NSValue 类似。

14.3　小结

本章介绍的几个类可以让非标准的 Objective-C 数据处理变得更简单。处理集合时特别方便，可以在第 13 章中看到，这些方法只适用于 Objective-C 对象。这些类提供了底层数据的一个简单的包装器，这样就可以在集合类中使用它们并对它们充分利用 Objective-C 面向对象的功能。我们生活在一个充满边界条件和遗留数据的世界。Objective-C 提供了一个最干净、最完整的接口。由于它源于 C 语言，所以它尤为适合处理这些任务。

第 15 章 处理时间和日期

本章概要

- 使用 NSDate 和 NSCalendar 处理日期
- 使用时间间隔
- 日期本地化

在计算机上处理日期一直以来都是很复杂的任务。日期可不像看起来的那么简单。其中有很多异常和边界条件，比如闰年、日历变化等。一个关于该主题的全面考虑表明，即使考虑到了这些异常情况，还有很多问题，例如日历的起始日期是什么时候、如何处理边界之外的日期等。我们所熟悉的 CE/BCE 标准实际上是一个非常低效的“拼凑”。

如果需要更多证据来证明日期处理有多么困难，可以回忆本世纪初在计算行业中发生的千年虫问题。天真的程序员开始想着可以用两位数来表示年份。但当世纪更替时，上百万行的代码都要重写。

即使是现在我们仍然面临着未来的日期问题，因为实际上大多数计算机通过 32 位整数将日期存储成从 1970 年 1 月 1 日开始的秒数。遗憾的是，这个计数器在 2032 年就会回滚。尽管这看起来还很遥远，但我还是要提醒你，在编写两位年份处理代码时，那些程序员也是同样看待 2000 年的。

即使你忽视了这些较大的问题，如何在应用中合理处理日期和时间还是很现实的问题。比如，如何确定以小时计的一周时间。你的第一反应可能就是将一天中的小时数乘以 7。这会是很常见、但很天真的反应。如果其中某一天从夏时制转到标准时间或者执行相反转换又如何呢？这样你的计算马上就变得不正确了。

在软件开发中这些问题很常见，并会导致由于公共关系受损以及客户问题而损失数百万美元的很严重的 bug。不要做那样的程序员。

本章将介绍 NSDate 类，用于构建和操作应用中的日期对象。接着会介绍 NSCalendar 类，该类支持指定计算日期时使用的规则。最后我会介绍 NSDateFormatter 类，它用于将一个日期值转换成可以显示给最终用户的表示。这 3 个类一起使用时就会成为满足一切日期处理需求的高效工具集。在 Objective-C 应用中处理日期时应优先使用它们。

构建日期

NSDate 是一个封装了某一给定时刻的类。它包括日期和时间。通过使用类方法+date 创建一个新的 NSDate 对象可以用于表示当前时间，或者通过 NSTimeInterval 创建一个 NSDate 对象表示将来或者过去的任意时间。代码清单 15-1 显示了如何创建一个表示当前时间的 NSDate 对象。它实际上使用两个不同的方法，第一种是+date 工厂方法，第二种则是使用了标准初始化函数的方法。

代码清单 15-1　创建一个 NSDate 对象

```
NSDate *now = [NSDate date];
NSDate *alsoNow = [[NSDate alloc] init];
```

使用时间间隔

NSTimeInterval 表示以秒为单位的时间片。通过它，就可以创建相对于其他日期的一个日期。比如，可以使用-initWithTimeIntervalSinceNow:这一初始化函数并传入 30 分钟的秒数作为参数，来创建一个表示“从现在开始的 30 分钟”的 NSDate 对象。

为了表示未来的时间测量，现在到将来的 NSTimeInterval 可以通过正整数来表示。换句话说，5 秒后可以通过值为 5 的 NSTimeInterval 表示。同样，要表示过去的时间，可以用负整数作为 NSTimeInterval 的值。这样，为了表示 5 秒之前的时间就可以创建值为–5 的 NSTimeInterval。相对于其他日期，可以通过为某一日期增加一个正的或负的 NSTimeInterval 来操作和创建一个相对于该日期的新日期，如代码清单 15-2 所示。

代码清单 15-2　通过时间间隔创建日期

```
NSDate *now = [NSDate date];
NSDate *anHourAgo = [now dateByAddingTimeInterval:-3600];
NSDate *anHourFromNow = [now dateByAddingTimeInterval:3600];
```

日期比较

你可以比较日期来确定哪个日期在前、哪个在后、是否相同，或者确定两个日期之间的时间间隔。

两个日期之间的时间差可以通过-timeIntervalSinceDate:方法来计算。你可以在日期上调用该方法，并传入另外一个日期作为参数。该方法返回两个日期之间的时间间隔。和创建一个新的 NSDate 对象一样，如果消息的接收方在给定日期参数之后，那么返回的 NSTimeInterval 是正数，如果在给定日期之前，则是负数。还有一个快捷的方法-timeIntervalSinceNow，该方法返回消息接收方的日期和当前时间之间的时间间隔。代码清单 15-3 显示了使用这些方法的一些示例。

代码清单 15-3　计算两个日期之间的时间差

```
NSDate *now = [NSDate date];
NSDate *anHourAgo = [now dateByAddingTimeInterval:-3600];
NSTimeInterval timeBetween = [now timeIntervalSinceDate:anHourAgo]; // 3600
```

此外，NSDate 类还提供了-laterDate:、-earlierDate:和-compare:方法来，用于比较日期。比较两个日期时，-laterDate:和-earlierDate:分别返回相对较早和较晚的日期。同时，-compare:方法返回一个标准的 NSCompareResult 结果，在日期排序时很有用。代码清单 15-4 就显示了这些方法的用法。

代码清单 15-4　比较日期

```
NSDate *now = [NSDate date];
NSDate *anHourAgo = [now dateByAddingTimeInterval:-3600];
assert([now laterDate:anHourAgo] == now); //真
assert([now earlierDate:anHourAgo] == anHourAgo); //真
assert([now compare:anHourAgo] == NSOrderedDescending); //真
```

使用NSCalender

尽管通过 NSTimeInterval 创建一个具体时间的 NSDate 实例很有用，但更多时候你希望创建具体某天或者基于日历而不是秒数的相对时间的 NSDate 实例。这不仅概念上容易想象，而且可以更精确并且不容易出错。在某些情况下，日历操作中的边缘条件很可能会“光顾”你。

Foundation 框架为此提供了 NSCalendar 类。它提供了一种通过更自然的日期组成，比如日、月、星期等，来指定日期的机制。这不仅适用于我们今天所使用的公历，还适用于一些专门的日历，比如希伯来日历、伊斯兰日历、佛教日历等。通过这种方式，它还提供了强大的本地化工具来为用户提供一个丰富的本地化体验。

要创建一个表示给定月份中的某天的 NSDate 对象，首先需要创建一个 NSDateComponents 对象并设置想包含的任何参数。你可以为日历创建一个 NSCalendar 对象，用于创建一个日期。两者配合使用，就可以创建一个 NSDate 对象来表示你期待的某天。代码清单 15-5 显示了实现过程。

代码清单 15-5　通过 NSDateComponent 和 NSCalender 来创建一个 NSDate 对象

```
NSDateComponents *components = [[NSDateComponents alloc] init];
[components setMonth:4];
[components setDay:13];
[components setYear:2010];

NSCalendar *currentCalendar = [NSCalendar currentCalendar];
NSDate *date = [currentCalendar dateFromComponents]; // 04/13/2010
```

同样，可以按照代码清单 15-6 所示创建一个“一周前”的日期。

代码清单 15-6 使用相对日期

```
NSCalendar *calendar = [NSCalendar currentCalendar];
NSDateComponents *components = [calendar components:(NSYearCalendarUnit |
                                                     NSMonthCalendarUnit |
                                                     NSDayCalendarUnit)
                                           fromDate:today];
[components setWeek:([components week] - 1)];
NSDate *oneWeekAgo = [calendar dateFromComponents:components];
```

甚至可以将给定日期从一个日历转换到另一个日历，方法是将一个 NSCalender 中创建的 NSDate 实例传入到另一个。代码清单 15-7 显示了如何实现。

代码清单 15-7 日期在不同日历间的转换

```
NSDate *today = [NSDate date];
NSCalendar *calendar = [NSCalendar currentCalendar];
NSDateComponents *components = [calendar components:(NSYearCalendarUnit |
                                                     NSMonthCalendarUnit |
                                                     NSDayCalendarUnit)
                                           fromDate:today];
NSCalendar *japaneseCalendar =
[[NSCalendar alloc] initWithCalendarIdentifier:NSJapaneseCalendar];
NSDate *inJapan = [calendar dateFromComponents:components];
```

使用这些技术，日期的创建会考虑日历的所有特性。比如，它会自动为你处理闰年和夏时制。

使用时区

处理日期和时间时经常遇到的另一个问题就是时区。Foundation 框架为此提供了 NSTimeZone 来指定给定地区日历对象的时区。和指定不同类型的日历一样，给定 NSCalendar 对象的时区影响到给定时刻与其他时区中的同一时间相比的计算得到的时间。换句话说，其他时区一周前的该时刻的小时数和你当前时区中相同时刻的小时数是不同的。

NSTimeZone 还通过类方法+knownTimeZoneNames 提供了所有时区的列表，你可以使用该类方法向用户呈现时区列表。

NSTimeZone 有一个类方法+knownTimeZoneNames，它可以列出它知道的所有时区。可以通过该类方法向用户呈现时区列表。

你可以通过+timeZoneWithName:工厂方法并指定时区名作为参数或者使用+timeZone-WithAbbreviation:工厂方法并指定时区的缩写来创建一个 NSTimeZone 对象，如代码清单 15-8 所示。

代码清单 15-8 创建一个 NSTimeZone 对象

```
NSTimeZone *est = [NSTimeZone timeZoneWithAbbreviation:@"PST"];

NSTimeZone *azZone = [NSTimeZone timeZoneWithName:@"America/Arizona/Phoenix"];
```

创建这些对象后，可以和 NSCalender 对象一起使用。如果没有在日历上显式地设置时区，就会使用系统的默认时区。如果要将时间设置成一个特定时区，可以设置 NSCalender 的时区，任何从日历中得到的日期都会进行相应的调整。

15.1　使用**NSDateFormatter**

多数情况下在处理日期时，最终还是需要将日期转换成向用户呈现的字符串。和处理日期本身一样，将日期转换成字符串时也有很多需要考虑的边界条件。除了简单的标准本地化问题，比如获取用户区域正确的月、日名称，需要考虑表示不同日期所使用的不同格式。比如，一个星期的某一天全名是 Tuesday，缩写后就成为 Thu 或者一个字母 T。月份就更复杂。它们可以用全名 September 或者缩写 Sept 或者一个数字 9 表示。在美国日期通常表示成 MM/DD/YYYY，而在欧洲通常表示成 DD/MM/YYYY。你可以看出，可能的差别是无限的。

为了处理格式化日期格式中的各种复杂性，Foundation 提供了一个 NSDateFormatter 类。利用该类你可以指定所需的任何类型的行为，并将指定的 NSDate 对象转换成与所需行为匹配的日期的相应字符串表示。比如，使用短的纯数字风格来显示日期，如 09/20/10，可以使用 NSDateFormatterShortStyle 来指定，如代码清单 15-9 所示。

代码清单 15-9　短数字格式日期的格式化

```
NSDate *date = [NSDate date];
NSDateFormatter *f = [[NSDateFormatter alloc] init];
[f setDateStyle:NSDateFormatterShortStyle];
NSString *dateStr = [f stringFromDate:date]; //输出 MM/DD/YY
```

这是双向的。也可以使用 NSDateFormatter 对象将一个表示给定日期的自然语言字符串转换成实际的 NSDate 对象。示例如代码清单 15-10 所示。

代码清单 15-10　将自然语言字符串转换成 NSDate 对象

```
NSDateFormatter *f = [[NSDateFormatter alloc] init];
[f setDateStyle:NSDateFormatterShortStyle];
NSDate *date = [f dateFromString:@"02/25/10"];
```

为任意日期格式创建一个 NSDateFormatter 对象是不可能的，通常会使用系统提供的标准格式中的一种。

15.2　小结

使用日期是一个复杂的主题，并且有很多细节对于粗心的程序员来说是陷阱。几年来，苹果和其他公司都在 NSDate 和 NSCalender 类上花了很多心思，尽量使日期处理不再是开发者的难题。你应该使用这些类，而不是手动处理日期。这样处理可以节省时间。

Part 4

第四部分

高级主题

本部分内容

第 16 章 通过多个线程实现多处理

16

本章概要

- ❑ 理解线程错误的根源
- ❑ 学习如何通过锁防止线程错误
- ❑ 使用`@synchronize`
- ❑ 通过 `NSThread` 创建线程
- ❑ 使用 `NSOperation` 和 `NSOperationQueue`

线程恐怕是最让有经验的程序员心惊胆寒的计算机主题了。应该是这样的。使用多个线程会导致很难定位、重现、修改的过程令人抓狂的 bug。同时，它们比任何现代软件技术都更能充分利用多核计算能力。这两种特性使得它们成为一个复杂的主题，因此值得用完整的一章来说明如何合理处理多线程。

每个程序至少有一个线程，通过称为主线程。在没有显式创建另一个线程的情况下，主线程从主函数开始执行然后负责执行剩下部分的应用代码。

从概念上讲，你可以将线程想象成指令按顺序执行的应用的一行代码的执行。创建另外一个线程时，实际上在应用中就有两个并行运行的独立执行线程。如果应用在单核、单个 CPU 的机器上运行，线程虽看起来是并发运行的，但是实际上它们会得到 CPU 分配的不同时间片。另一方面，如果应用是在多核或者多 CPU 机器上运行，在应用中两个线程就很有可能是同时执行的。

当两个线程通过这种方式并发执行时，很有可能两个线程会同时试图访问相同的内存块。如果这发生了，具体行为是不确定的，这会导致程序出现错误。这种情况称作不安全的线程状态。只有在编写所执行的代码时没有考虑线程安全的情况下，这才会发生。

为了防止这些情况，必须防止一个线程在同一时间访问另一个线程正在访问的相同内存或数据。这称作让代码变得*线程安全*。重要的是使用多个线程的任何时候都必须确保所写的代码是线程安全的。在多线程中调试很困难，因为通常这样的 bug 只有在极少的情况下发生。这就意味着 bug 可能因为计算机的速度和配置而在特定用户的计算机上出现，但永远不会在自己的计算机上出现。此外，由于调试器可以在某一时刻停止应用的运行，并且它可以将应用的执行限制到具体的某个线程，因此线程处理 bug 通常不会在调试器中显示。因此，线程处理 bug 通常也被称为“海森堡虫子”。得到这样的绰号是因为你知道它们会发生，但试图观察时它们就消失了。

很多开发者看到 Objective-C 中所有可用的优秀的线程相关工具时，第一反应就是多线程一定可以通过某种方法解决所有的设计问题。很容易为不同任务在后台启用一个线程，从而让主线程仅仅处理用户界面交互。但是这样做很糟。即使对于网络代码，其他语言有 99%的可能会鼓励你使用多线程来防止阻塞 I/O，但对于 Objective-C 来说，线程可能是一个错误的工具。多线程是一种仅适用于少数需求的技术，比如在进行 CPU 密集型计算时。即使在这些情况下，合理编写的线程安全的代码也可能被 GUI 交互等共享资源阻塞。这些问题可能使多线程代码在某些情况下比单线程处理还要慢。

Objective-C 提供了很多创建和控制线程以及编写线程安全代码的工具。本章会介绍其中一些编写线程安全代码所需的关键技术，然后介绍一些使得 Objective-C 的线程创建和使用变得简单的类。



警告

需要注意的一点是，Cocoa UI 框架（UIKit 和 AppKit）实际上不是线程安全的。任何时候同应用的任何 GUI 元素的所有交互都必须在主线程进行。如果你必须从后台线程更新一个 GUI 元素，那么必须在主线程上调用一个 `NSObject` 方法`-performSelectorOnMainThread:withObject:...`来实现。



警告

一些程序员遇到的一个常见 bug 出现在使用 `NSNotificationCenter` 的通知时。通知可以在发布它们的线程上发送。这就意味着如果从后台线程发送通知来更新 GUI 组件，就有可能出现线程安全问题。因此，在使用线程的应用中要小心使用通知。

16.1　同步代码

编写线程安全代码的要点就是要记住，没有线程可以安全地读写可能正被另一个线程读写的特定内存块。如果被读写的内存同时也会被底层线程改变，应用的行为就不确定。任何事情都可能发生。因此关键就是要确保在写入某个特定内存块或者变量时没有其他线程可以在其完成前进行读取。

在写入时阻止另外一个线程访问正被写入的内存的最常见方式就是使用互斥锁，也称为互斥体。

16.1.1　使用锁

互斥锁，也称为*互斥体*，是一种可以防止同时访问同一资源（通常是内存）的对象。之所以称为互斥锁是因为它会锁住对资源的访问，并使得在某一时刻可以独占访问一个线程。

Objective-C Foundation 框架提供了两种主要的互斥锁。第一种是简单的 `NSLock` 类。`NSLock`

表示一种你可以实例化然后可以在写入一个特定变量或者内存位置时进行锁定的简单互斥体。其他想读写相同变量的线程必须在开始数据访问之前，试图锁定同一个 NSLock 对象。试图锁定 NSLock 对象会阻塞线程直至锁被解锁，然后也可能被重新锁定。这样，试图锁定 NSLock 对象就可以在第一个线程结束访问前防止其他对数据的访问。

要创建一个 NSLock 的实例，可以使用标准的分配/初始化模式进行实例化。代码清单 16-1 显示了一个实现的示例。通常，需要将 NSLock 的实例作为任何访问数据的类的成员变量保存。将 NSLock 作为变量维护并让两个线程都可以访问到很重要，这样它们就可以相应地获取到锁。

代码清单 16-1　创建 NSLock 的一个实例

```
-(id)init
{
    if((self = [super init]))
    {
        lock = [[NSLock alloc] init];
    }
    return self;
}
```

通常，你可以按照上述方法利用类的初始化函数创建 NSLock 的一个实例。在数据的存取器函数中，通常需要在试图访问数据前锁定 NSLock 实例。代码清单 16-2 显示了使用了该技术编写的一些存取器函数的示例。

代码清单 16-2　存取器函数示例

```
-(void)setSomeVar:(id)inValue
{
    [inValue retain];
    [lock lock];
    id originalValue = someVar;
    someVar = inValue;
    [lock unlock];
    [originalValue release];
}

-(id)someVar
{
    id ret = nil;
    [lock lock];
    ret = [someVar retain];
    [lock unlock];
    return [ret autorelease];
}
```

记住试图锁定一个已经被锁定的 NSLock 实例会导致线程阻塞直至获得该锁。如果这会造成问题，NSLock 提供了两个可以帮助你的便捷方法。第一个是-tryLock 方法。该方法会尝试获取一个锁，如果不能，它会立刻返回 NO。如果可以锁定锁，就返回 YES。这在你想在执行一些操作之前试图获取一个锁的情况下很方便，但是如果无法获取锁，你可以在等待再次尝试的同时

执行一些其他操作。

NSLock 提供的第二个便捷方法就是-lockBeforeDate:方法。和-tryLock 方法一样，该方法会返回一个布尔类型的 YES 或 NO，取决于是否可以锁定 NSLock 的实例。在本例中，该方法会阻塞一段时间直至到指定时间。如果该方法过期，还是无法获取锁，就会返回 NO。

NSLock 的最大问题就是如果错误地去尝试锁一个已经被所在线程锁住的锁，结果就会发生熟知的“死锁”。因为尝试锁定该锁会阻塞当前线程，这就会导致你等待一个锁被解锁，但是它永远不可能被解锁，因为负责解锁的线程就是等待它被解锁的线程。这听起来及其扭曲复杂，但在复杂应用中这的确会发生。

为了解决该问题，还有另一种称作 NSRecursiveLock 的锁。该锁记录锁定它的线程，如果该线程再次试图锁定它，就会立刻返回。你不需要担心访问当前线程正在锁定的数据。因此，试图锁定一个已经被锁的锁没有任何意义。在这些情况下使用一个 NSRecursiveLock 实例就可以解决这些问题。

16.1.2　使用@synchronize关键字

使用锁是确保代码是线程安全的一种简单高效的方式。但是，如果写过一定量的基于 NSLock 的代码后，就会发现一些模式。

第一种模式就是你通常锁定对具体变量或者一个特定对象的所有成员变量的访问，这样就会有和这些具体变量相关联的锁的实例。通常会有一个使用给定类的成员作为其锁对象的锁。可以在想访问该类的数据时锁定该锁。另外，你可以有和特定变量关联的特定锁，并希望确保访问那些特定对象时会锁定那些锁。换句话说，你试图保护数据之间的耦合。理想情况下，你期望着有些语言结构可以在代码中表现这种关系。

第二种可能出现的模式就是在处理线程锁时实际上很容易忘记解锁。如果发生了这种情况，就会遇到死锁。在有异常的情况下问题就更大了，这会导致普通的调用栈的执行被打断。由于这些问题，Objective-C 引入了一个内置的语言指令，称作@synchronized。该指令提供了一个包括了特定作用域和变量参数的内置的底层互斥锁机制。这就意味着@synchronized 指令可以为一个特定变量指定锁并让该锁在特定的代码域存在。代码清单 16-3 显示了实际使用@synchronized 指令的示例。

代码清单 16-3　使用@synchronized

```
-(void)setSomeVar:(id)inValue
{
    [inValue retain];
    @synchronized(someVar)
    {
        id originalValue = someVar;
        someVar = inValue;
        [originalValue release];
    }
}
```

可以看出，@synchronized 指令接收单个参数，指定该锁的目标变量。此外，它还接收了一个代码块，在大括号中指定，这指定了被锁定的域。本质上，你可以将此看做作用域锁。该锁仅仅在大括号内的代码域中存在。

通常，@synchronized 指令通常和 self 变量一起使用来指定整个对象都在@synchronized 块的作用域中被锁定。代码清单 16-4 就是一个示例。

代码清单 16-4　使用 self 作为同步变量

```
-(void)setSomeVar:(id)inValue
{
    [inValue retain];
    @synchronized(self)
    {
        id originalValue = someVar;
        someVar = inValue;
        [originalValue release];
    }
}
```

@synchronized 的一个优点就是因为它指定了锁的作用域，如果发生了异常之类的情况导致它退出了该作用域，锁就会被释放。

使用@synchronized 而不是 NSLock 或 NSRecursiveLock，被视为一种确保线程安全的更现代、更安全的方式。可能的情况下，应在应用中使用该技术。



说明

不可变的 Foundation 类，如 NSString、NSArray、NSDictionary 和 NSSet 等，由于创建后无法修改因此也自然被认定是线程安全的。但是存储它们的变量则不是，因此在修改时需要通过锁保护。

16.1.3　理解原子性

保证代码线程安全的另一个可用工具和属性的使用相关。atomic 属性标志指定无论多少个线程访问给定属性，其值的设置或获取，都会得到一个“完整”的值，而不是部分值。本质上，它确保@synthesize 指令为你的属性所创建的存取器函数在附值或获取之前，会在生成的存取器函数中利用一个@synchronized(self)代码块。当你指定 nonatomic 标志时，就不会使用@synchronized 代码块。

通过指定 atomic 标志（这是默认的），你可以指定属性存取器函数本身是线程安全的。也就是说，如果两个线程同时通过属性存取器函数访问某个特定的成员变量，那么该操作是线程安全的。然而，这不能确保整个对象或者对这个对象的多个不同的存取器函数的不同调用是线程安全的。为此，需要实现某种形式的对象范围的锁。

16.2 创建 NSTread

在 Objective-C 中有几种创建新线程的方式。第一种就是使用 NSThread 对象。

16.2.1 创建线程

要使用 NSThread 类创建一个线程，可以使用工厂方+detachNewThreadSelector:toTarget:withObject:，或者直接使用标准的初始化函数-initWithTarget:selector:object:。在前一种方法中，会创建并启动一个线程，并运行选择器和目标提供的代码。在后一种方法中，线程会被初始化，但不会实际运行直至调用了-start 方法。代码清单 16-5 显示了通过工厂方法创建一个线程的示例。

代码清单 16-5 创建一个新线程

```
[NSThread detachNewThreadSelector:@selector(work:)
                           toTarget:self withObject:someData];
```

在本例中，所调用的选择器是在当前对象上定义的-work:方法，通过该对象调用该方法时使用 self。-work:方法接收一个参数。我们通过 someData 传入该参数。该代码的功能就是创建一个新线程，在该线程中会使用 self 调用-work:方法，并将 someData 参数传入。线程启动后，该方法就会返回。

16.2.2 控制运行的线程

一个线程创建和分离后，它就会继续运行直至用于启动线程的选择器退出。如果需要控制线程的停止，通常需要在选择器的运行循环内，为在主线程中设置的某个变量包含一个检查机制。换句话说，如果你有一个在后台持续运行的某任务，直至用户单击停止按钮它才停止运行，你就需要一个可以在前台线程设置，并在后台线程检查的变量。示例如代码清单 16-6 所示。

代码清单 16-6 常见的后台线程运行循环

```
-(void)work:(NSDictionary *)somData
{
    while([self continueRunning])
    {
        //执行一些操作

        [self doSomethingWith:someData];
    }
}
```

在本例中，有一种比较暴力的技术。在该方法之外没有任何操作，对运行循环也没有做什么特殊处理。

在某些极少的情况下，你可能希望允许当前线程的运行循环在你的线程内实际得到一些处理

时间。比如，有些类（如 NSURLConnection）可以安排在当前运行循环上运行，而不是在各自的线程运行。

为了确保这些类和方法在运行循环上得到合理的时间，你需要确保给该运行循环一个在你的线程运行循环中运行的机会。代码清单 16-7 显示了后台线程的另一个示例，它实际上也给了当前线程运行循环一个运行的机会，以执行需要的操作。

代码清单 16-7　一个使得当前线程运行循环有机会运行的运行循环

```
-(void)work:(NSDictionary *)somData
{
    while([self continueRunning])
    {
        //执行一些操作

        [self doSomethingWith:someData];
        [[NSRunLoop currentRunLoop] runUntilDate[NSDate date]];
    }
}
```

你不需要给它很多运行时间，只要简单地调用 runUntilDate 并传入现在日期。如果需要处理任何东西，会给你机会。指定一个将来的日期仅会让当前线程在某个位置阻塞直到那个时期，运行循环才开始在后台运行。如果在当前运行循环中不需要做任何事，它就会睡眠而不干任何事情。



说明

代码清单 16-6 和代码清单 16-7 中的一个假设就是 continueRunning 属性声明成了原子性的。这就确保在该线程和主线程设置和获取该值是线程安全的。

16.2.3　访问主线程

之前提到 Cocoa GUI 框架不是线程安全的。如果你在后台线程中需要访问某个 GUI 元素，就需要通过主线程来实现。不能在后台线程实现。显然，如果不能在后台线程中访问主线程将会是很大的一个限制。幸好，Objective-C 提供了在后台线程轻松访问主线程的方法。

可以想象你在后台线程需要计算一些值，然后更新 GUI 控件来显示计算得到的值。为此，在后台线程你仅需使用 NSObject 方法-performSelectorOnMainThread:withObject:waitUntilDone:。该方法接收 3 个参数，第一个参数是要调用的选择器名，第二个参数是一个可以传入到要调用的方法的可选对象。最后，第三个参数指定是否希望阻塞当前线程，直到在主线程上调用的方法结束。代码清单 16-8 就显示了该方法实际应用的示例。

代码清单 16-8 从后台线程更新 GUI 控件

```
-(void)doSomethingWith:(NSDictionary *)someData
{
    NSValue *calculatedValue =
            [someObject calculateValueFromData:someData];
    [self performSectorOnMainThread:@selector(updateGui:)
                        withObject:calculatedValue
                     waitUntilDone:NO];
}
```

16.2.4 通过执行选择器跨线程

除了在当前上下文中让主线程执行特定操作外，可能还有从主线程到后台线程的通信需求。类似更新 GUI 控件或同主线程通信，`NSObject` 也提供了一种在指定的后台线程上执行选择器的方法。该方法就是`-performSelector:onThread:waitIntilDone:`。这和之前的方法的工作原理一样，只不过不是在主线程上执行选择器，而是在指定线程上执行选择器。

16.3 使用 NSOperation 和 NSOperationQueue

`NSThread` 是一个很强大的类，它是一种在底层创建和管理线程的一个很好的方式。但是，如果考虑到创新，`NSThread` 使用的技术本质上与过去 40 年来创建线程的技术相同。最近，苹果添加了一些新的线程处理功能。

通过这种方式管理线程可以很困难并易于发生错误。手动管理线程的一个最复杂的方面就是应用的线程的最佳线程数目取决于目前有多少个系统线程在运行以及所运行的机器有几个内核。在理想情况下，可以生成合适数量的线程来充分利用 100%的 CPU 资源。程序员很难知道实现该目标要用几个线程。最近，苹果添加了一些新的线程处理功能到 Objective-C 来处理这种困境。新的线程模型的核心就称作 GCD。

GCD 以 `NSOperation` 和 `NSOperationQueue` 为核心。这两个类一起提供了一个把线程当作单独的原子性任务的高层次面向对象的抽象。

这套类中的核心类是 `NSOperation`。`NSOperation` 提供了一个你可以继承的基类，用于定义一个可以在后台线程执行的任务。你可以将 `NSOperation` 对象视为要执行的任务实例。你可以继承 `NSOperation` 类并创建一个自定义的操作类。然后实例化该自定义类，并将该操作其交由一个负责管理操作的 `NSOperationQueue`。`NSOperationQueue` 甚至会生成在后台处理任务的合适数量的线程。`NSOperationQueue` 在幕后利用了 GCD 来调度和启动合适数量的后台线程来处理你提交给它的所有的操作。可以配置操作依赖，这样一来一个给定的操作只能在它所依赖的所有操作都完成后才能开始。此外，队列还可以配置成并发运行或者顺序执行这些操作。



前后参照

第 5 章介绍了一些可以用于同 GCD 交互的低层次的 Objective-C 函数。



说明

Objective-C 的开发新手可能会错误地认为 NSOperationQueue 是一个用于将任务在当前线程上排队的有用工具。这是不准确的。NSOperationQueue 实际上和后台进程相关。这是默认的操作。

16.3.1 创建操作

NSOperation 和 NSOperationQueue 最适合“易并行”的任务。它们是没有实际依赖，但包含受 CPU 限制的计算。这些任务会在多核 CPU 上高效运行。试想一下对一系列图片进行某种图形效果处理的应用示例。要创建一个可以实现该任务的 NSOperation，首先需要派生 NSOperation 类以创建一个自定义操作类，如代码清单 16-9 所示。

代码清单 16-9 一个自定义的 NSOperation 子类

```
@interface PhotoBlurOperation : NSOperation
{
    NSImage *photo;
    NSString *photoPath;
}

-(id)initWithImageAtPath:(NSString *)pathToImage;
-(void)blur;

@end

@implementation PhotoBlurOperation

...

-(void)main
{
    if(![self cancelled])
         photo = [NSImage imageAtPath:photoPath];
    if(![self cancelled] && [self photo])
        [self blur];
    if(![self cancelled])
        [photo writeToOutputPath:...];
}

@end
```

这是一个极其简单的示例，显然在这里我没有给出模糊处理的细节，但无论操作如何复杂，基本都是一样的。理解的要点就是操作的主要入口和出口都是在 NSOperation 子类中实现的 main 方法。这就是创建处理操作的线程开始执行的地方，在 main 方法退出时线程也会退出。线程不可能被强制终止，如果要取消 NSOperation，那要取消属性必须设置成 YES。在 main 方法的实现中以及在操作中的任何耗时方法中，都需要周期性地检查 cancelled 属性来确认 NSOperation 是否被取消。如果被取消，你就需要清理已经开始的任何工作并尝试尽快退出 main 方法。例如在代码清单 16-9 中，就需要在 blur 方法中定期检查 cancelled 属性，也可能在图片的各个部分循环时进行检查。

16.3.2 将操作加入到队列

在创建了自定义操作的实例后，就可以将这些操作加入到一个 NSOperationQueue 中。NSOperationQueue 会管理所生成的线程，这些线程负责你提供给它的操作。

下面是图片模糊处理的示例，假想有一个图片目录需要模糊处理。你就需要从目录加载所有的文件，为其中的每个图片创建一个模糊处理的操作，然后将操作加入到 NSOperationQueue 来真正执行模糊处理。代码清单 16-10 就是将操作加入队列的一个示例。

代码清单 16-10 将其操作加入到队列

```
-(void)processDirectoryOfFiles:(NSString *)inDirectoryPath
{
    NSArray *filesInDirectory = [...];  //获取文件列表

    queue = [[NSOperationQueue alloc] init];

    for(NSString *imagePath in filesInDirectory)
    {
        PhotoBlurOperation *op = [[PhotoBlurOperation alloc]
                                  initWithImageAtPath:imagePath];
        [queue addOperation:op];
    }
}
```

操作加入到队列后，在执行完以及被取消之前都会在队列中。

16.3.3 控制队列参数

在每个操作加入队列后，队列会不断取出操作进行处理。NSOperationQueue 和 GCD 相配合就可以基于现有系统状态自动配置合适数目的线程来处理操作。如果你想手动配置要使用的线程数目，你可以使用 maxConcurrentOperationCount 属性。设置该属性可以限制队列使用的线程数目。如果将其配置成 1，实际上就会创建一个串行处理的队列。

NSOperationQueue 提供了确定队列状态的其他方法。你可以通过-operationCount 方法确定队列中等待执行的操作数目。还可以通过方法访问操作本身。如果想要在队列完成所有操

作之前阻塞当前线程，可以使用`-waitUntilAllOperationsAreFinished`方法。

16.3.4　使用不同的操作

除了可以从 `NSOperation` 继承来创建自定义操作外，Foundation 还提供了两个自带的 `NSOperation` 子类。`NSInvocationOperation` 用于创建一个调用现有对象上给定方法的操作，`NSBlockOperation` 接收一个作为操作主函数的一部分被执行的代码块。

对于已经有实现所需任务的代码并且该代码是现有类的一部分的情况，使用 `NSInvocationOperation` 很方便。重构代码到 `NSOperation` 就不方便了。`NSInvocationOperation` 支持原地调用一个给定的原有方法。例如，假想你有一个 `NSImage` 类别用于模糊处理操作。你可以通过代码清单 16-11 所示的代码创建一个实现类似功能的模糊处理的操作。

代码清单 16-11　使用 `NSInvocationOperation`

```
-(void)processDirectoryOfFiles:(NSString *)inDirectoryPath
{
    NSArray *filesInDirectory = [...]; //获取文件列表

    queue = [[NSOperationQueue alloc] init];

    for(NSString *imagePath in filesInDirectory)
    {
        NSImage *image = [NSImage imageAtPath:...];
        NSOperation *op = [[NSInvocationOperation alloc]
                            initWithTarget:image
                            selector:@selector(blur)
                            [object:nil];
        [queue addOperation:op];
    }
}
```



警告

在本例中，因为需要提前加载所有图片，这会占用很多内存。基于此原因，该示例是一个坏例子，不应该照搬照抄。

目的就是为了展示如何使用`NSInocationOperation`。

可以看出，可以创建一个 `NSInvocationOperation` 并可以让它调用图片对象的模糊处理，之后将其加入到操作队列，就像 `NSOperation` 子类一样。

如果想要执行的操作可以通过代码块表示，那么 `NSBlockOperation` 就很方便。代码清单 16-12 显示了使用 `NSBlockOperation` 进行相同的模糊处理操作。

代码清单 16-12 使用 NSBlockOperation

```
-(void)processDirectoryOfFiles:(NSString *)inDirectoryPath
{
    NSArray *filesInDirectory = [...]; //获取文件列表

    queue = [[NSOperationQueue alloc] init];

    for(NSString *imagePath in filesInDirectory)
    {
        NSOperation *op = [NSBlockOperation blockOperationWithBlock:^
              {
                 NSImage *image = [NSImage imageAtPath:imagePath];
                 [image blur];
              }];

        [queue addOperation:op];
    }
}
```

如果需要可以通过`-addExecutionBlock:`方法将多个执行代码块加入到单个 `NSBlockOperation` 实例中。该操作可以顺序执行每个代码块并在所有代码块都执行后才视为完成。



警告

有些接收代码块参数的调用会自动在后台线程执行代码块。这不都会有文档记录。所以在将任何代码块传入到一个不是自己编写的 API 时，必须假定它在另一个线程执行，所以必须在编写时考虑线程安全。这包括注意不要在需要访问的代码块外改变对象。

16.4 小结

本章中所展示的所有线程工具在创建一个充分利用多处理器和多内核机器的应用时会发挥强大的威力。能力越大，责任越大，你必须确保在使用这些工具时合理进行应用架构。

在可能的情况下，如果有选择，我推荐你在代码中使用单线程的设计，而不是引入多线程。只有很适合你的问题域时才使用多线程。

在使用多线程时，使用能够达到目标的最高层次的抽象，并且确保使用合适的线程锁和同步来防止多个线程在内存上相互竞争。本章提供了利用 Objective-C 编写高性能的多线程应用所需的所有工具。

第 17 章

Objective-C 设计模式

17

本章概要

- ❑ 理解 Objective-C 中设计模式的使用
- ❑ 学习如何利用 Objective-C 实现单例
- ❑ 使用委托来委托责任
- ❑ 利用通知观察变化

使用 Objective-C 和 Foundation 框架开发的乐趣之一就是，其设计者在考虑 API 和语言的设计时完全采用了最先进的软件开发方法。实际上，有些专家宣称这些方法其实都源于 Objective-C。

在 Objective-C 和 Foundation 中得到了很好体现的这些方法就是设计模式这种概念。实际上，很多我们在现代编程中使用的设计模式的首次实现都源于 Objective-C 社区。尽管其名称可能和现代的叫法不同，比如“职责链”、“观察者”等，这些常见（当前）的设计模式都在 Objective-C 语言中得到了很好的体现。

本章会展示如何利用 Objective-C 实现一些比较常见的设计模式。Objective-C、Foundation 和 Cocoa 在它们各自的 API 中大量使用设计模式，因此，你可能经常见到 Objective-C 实现的设计模式。了解 Objective-C 如何实现这些设计模式会很有帮助，因为在 Objective-C 等动态编译语言中的实现会和在 C++或 Java 等更严格的强类型语言中的实现稍微不同。

17.1 识别解决方案中的模式

你是否注意到随着时间推移，相同的问题会在不同项目的应用开发中不断出现呢？在你埋头工作时，可能面临一个和之前在其他地方处理的问题很接近但又不完全一样的编程环境。这两个问题可能类似，但是还没类似到可以完全复用代码的程度。设计模式是某个具体编程问题的通用化、可以复用的解决方案，可以在各种差别较大的应用架构中重用。有经验的开发人员会发现自己在不同的应用上下文中面临相同的常见问题。通常，利用现有的知识和代码，这些通用问题的解决方案可以在不同的上下文中应用。一般来说，没有一种可以从前一个方案中完全复用代码的情况，但是可以使用前一次解决问题时所使用的思想。

作为一个常见的设计模式的例子，试想一下有两个对象 `Foo` 和 `Bar`，`Bar` 想要在 `Foo` 有任

何变化的时候收到通知。应该如何解决这个问题呢？

在这种情况下你可能会想着让 `Foo` 保存 `Bar` 对象的引用，这样在任何目标事件发生时，它就知道要调用 `Bar` 对象上的一个方法来告诉 `Bar`。这是一个很常见的问题和解决方案。这两者合起来就构成了一种设计模式。事实上，该模式称为委托设计模式。`Bar` 就成为了对象 `Foo` 的委托。

开发人员认为在两种不同状态下比较合适使用某个特定的设计模式。第一个就是应用的设计阶段。学习设计模式会派上用场，因为这样一来，开发者在讨论关于应用设计的抽象思想时，有一种“共同语言”来讨论某个特定解决方案的细节。换句话说，作为开发人员你不需要描述给定解决方案的细节，而只需简单地说“在这里我们使用委托来处理该问题。”其他开发人员马上就知道你在说什么，并且在没有任何附加信息的情况下就可能会实现解决方案。这使得设计模式成为了专家级开发人员工具箱中的一个关键工具。

第二个设计模式通常出现的地方就是开发人员忙于处理一个特殊问题时。开发人员通常发现自己遇到了表面上需要复杂解决方案的疑难问题。

设计模式可以使你在遇到任何编程问题就会马上想到一个正确的解决方案。

基于这些原因，值得花时间详细研究设计模式。你应该尽可能利用和所用语言相关的资源来学习设计模式。换句话说，也就是要学习 Objective-C 和 Cocoa 的设计模式。但是，关于设计模式通用研究的大部分资源使用的都是伪代码，而不是特定于 Objective-C 的代码。不要被这个吓倒。这些资源的大部分都适用于 Objective-C。

在下面几节中，我会介绍如何使用 Objective-C 实现几个具体的设计模式。这不是 Objective-C 中所有设计模式的罗列，但足以让你上手并开始理解 Objective-C 在实现这些模式时和其他语言有什么不同。这里我并不打算介绍完整的模式类别，而是给出一些说明性的模式示例，以帮助你作为 Objective-C 程序员更好地把握其他设计模式。

17.2　用 Objective-C 描述设计模式

大多数设计模式的书籍都遵循一个具体的模式描述格式。通常，这些书开始都会指出一个特定类型的问题。然后在问题语句后紧跟着一个解决方案的描述，包括解决方案代码以及最后关于解决方案的讨论。考虑到将要讨论的设计模式，我也会遵循该模式。

17.2.1　使用单例

1. 问题

你需要确保应用中的一个特定类有且仅有一个实例，并为其提供一个全局访问点。原因可能会是设计约束或者为了控制对有限资源的访问。

2. 解决方案

该问题的解决方案就称作单例。单例是一个可以确保不会创建多于一个实例的类。通常，在第一次调用类的构造函数时就会创建单一的全局实例。接下来调用构造函数时会检查该全局实例

是否存在。如果存在就返回全局实例的引用而不是创建一个新的对象。

在 Objective-C 中实现单例通常包括几个步骤。首先必须创建一个全局实例。该全局实例通常存储在一个全局变量中。该全局实例的一个重要特点就是必须要将该实例设置成 nil。之后在调用初始化函数时候会检查该变量是否为 nil。代码清单 17-1 就显示了一个典型的全局单例实例的声明。通常，该声明放在想将其作为单例的类的实现文件中。

代码清单 17-1 全局实例定义

```
static MyClass *instance = nil;
```

在你创建了用于存储全局变量实例的变量后，你需要通过工厂方法提供一个对该实例的全局访问。该方法会检查实例是否存在，如果不存在就创建它，如代码清单 17-2 所示。

代码清单 17-2 单例工厂方法

```
+(MyClass *)sharedInstance;
{
    @synchronized(self)
    {
        if(!instance)
            [[self alloc] init];
    }
    return instance;
}
```

该方法被声明成类方法，因此通过类本身就可以访问到。在调用时首先检查实例变量是否初始化。如果没有，就初始化实例变量，最后返回全局实例。

全局实例变量的实际初始化在+allocWithZone:中进行，这是调用前一个+alloc 方法时最终调用的方法。该方法如代码清单 17-3 所示。

代码清单 17-3 初始化全局实例

```
+(id)allocWithZone:(NSZone *)inZone;
{
    @synchronized(self)
    {
        if(!instance)
        {
            instance = [super allocWithZone:inZone];
            return instance;
        }
    }
    return nil;
}
```

使用该方法而不是工厂方法来验证和初始化全局实例，确保了即使某些人试图利用标准的+alloc 和-init 方法来创建一个单例类的实例，也会收到全局实例而不是新的副本。

为了确保全局变量的安全性，还需要实现几个方法。比如，还需要实现-copyWithZone:

方法。该方法在对象收到调用-copy 方法的消息时会被调用。通常，这用于创建一个对象及其属性的副本。通过重写该方法，你可以简单地返回自身，换句话说，因为这只能在全局实例上调用，那么只能返回全局实例的指针。如何实现该方法如代码清单 17-4 所示。

代码清单 17-4　实现-copyWithZone:

```
-(id)copyWithZone:(NSZone *)inZone;
{
    return self;
}
```

在没有垃圾回收、保留计数的内存模型中，需要重写合适的方法来避免释放全局实例。因此，需要重写-retain、-retainCount、-release 和-autorelease 这几个方法。重写每个方法的目的都是为了避免对象被保留或者被释放。你只能有一个对象的实例，并且保留计数只能是 1。

代码清单 17-5 显示了需要在单例中实现的其中 4 个方法。

代码清单 17-5　实现内存管理方法

```
-(id)retain;
{
    return self;
}

-(unsigned)retainCount;
{
    return NSUIntegerMax;
}

-(void)release;
{
    // empty
}

-(id)autorelease;
{
    return self;
}
```

实际上每个被重写的方法什么都不做。实现了单例以后，使用时只要使用全局工厂方法即可。第一次访问时，全局实例会被初始化。所有接下来对工厂方法的访问都是返回最初的实例。因为重写了所有内存管理的方法，任何试图保留或者释放该对象都是徒劳。



说明

我将在这里描述的在 Objective-C 中实现单例的方法称为“安全”方法。换句话说，如果将这些代码发布给可能误用单例的第三方，这里展示的这些技术是最安全的。在该设计模式的讨论部分，我会展示一个便捷的“不安全”版本，是否使用取决于你。

3. 讨论

和 Objective-C 一起使用的 Foundation、Cocoa、Cocoa Touch 和其他很多框架都充分利用了单例。一般来说，这都发生在目标对象对仅有一个实例的资源的访问进行封装的场合。比如，`NSNotificationCenter` 的核心位置有一个单例。同样，在 Cocoa Touch 中由于只能运行一个应用，`UIApplication` 也是有一个单例。

单例是最强大也最常用的设计模式之一。经常在需要使用全局变量的场合中使用。与全局变量相比,单例的优势就是它能更好地控制所创建的全局实例,并且可以确保只能有一个全局实例。没有单例，应用中的某些部分就可能重新初始化一个全局实例。单例可以防止这些。

之前展示的创建单例的技术是一种“安全”的技术。如果你是一个供第三方使用的库的开发者，或者你所在的开发团队无法确定 API 使用者是否了解你的对象是单例，并且不需要被释放或者保留等，我推荐使用这种技术实现单例。

但是，如果你是为自己开发代码的独立开发者，或者你确信不会过早释放单例，就可参考代码清单 17-6 所示的单例模式版本。

代码清单 17-6　单例的简化版本

```
static MyClass *instance = nil;

+(id)sharedInstance;
{
    if(!instance)
    {
        instance = [MyClass new];
    }
    return instance;
}
```

这就是实现一个不安全单例所需要的全部代码。所有之前介绍的方法都是为了防止错用代码。这里少了重写内存管理方法、线程安全等。作为该单例的用户，需要知道的要点就是单例不能被释放或者保留。此外，需要确保在启动其他外部线程之前完成全局实例的初始化。尽管立刻从多个线程读取实例变量的内容是安全的，但是写入是不安全的。因此，必须确保对共享实例方法的第一次访问必须在创建任何可能需要访问该全局实例的线程之前完成。

如果确信这样就可以满足你的所有要求，这就是一个单例实现的更简单的版本。通常来说，这是我在代码中的使用的实现。

Objective-C 中这个设计模式的学习很有用，因为它在实现中使用了工厂方法。Objective-C 比任何其他语言使用了更多工厂方法。因此，使用工厂方法来访问单例对于大多数 Objective-C 开发者很自然。有时，在其他语言中，你需要特意阻止开发人员通过标准的构造函数创建一个对象的实例。你需要记录工厂方法，并在所有代码中发布警告以确保开发者可以使用工厂方法。大多数有经验的 Objective-C 开发者首选都会考虑用工厂方法来创建新对象，因为工厂方法比使用标准的初始化函数更方便。此外，如果你使用“安全”的单例实现，Objective-C 提供了足够的工具来防止粗心的开发者不小心释放单例并导致 bug。

17.2.2　委托责任

1. 问题

你有两个对象，一个需要在另一个状态变化时得到通知。另外，其中的一个对象希望在运行时让另一个对象负责确定行为变化。

2. 解决方案

该问题的解决方案就是委托模式。委托模式定义了一种解决方案，其中一个对象保存另一个对象的引用。被引用的对象实现了事先确定的接口，该接口用于将引用对象中发生的变化通知给被引用对象。该模式不仅可将引用对象中的变化通知给被引用对象，同时它也可以用于将责任委托给被引用对象，或在运行时代替引用对象进行决策。

比如，给定 `Foo` 和 `Bar` 两个对象，`Foo` 可能选择将责任委托给 `Bar`，以确定在错误发生时如何处理。当错误发生时，`Foo` 对象告诉它的委托 `Bar` 错误发生了，`Bar` 就有机会介入。

在 Objective-C 中实现委托模式包括创建一个定义委托接口的协议，创建一个实现了协议的委托对象，并在委托对象内包括一个被委托对象的引用。代码清单 17-7 显示了协议和委托对象实现的示例。在本例中，`MyClass` 是将责任委托给被委托对象的类。委托的协议称作 `MyClassDelegate`。

代码清单 17-7　委托协议以及委托类的接口

```
@protocol MyClassDelegate
-(void)requiredMethod;

@optional
-(void)somethingOptional;

@end

@interface MyClass
{
    id<MyClassDelegate> delegate;
}
@property (assign) delegate;
@end
```

可以看出，`MyClass` 实例保存一个委托的应用，委托需要实现 `MyClassDelegate` 协议。

为此，需要将 `MyClass` 的委托对象事先设置成一个实现了委托协议的对象实例。一个很重要的警告就是在 Objective-C 中使用委托时，委托必须指定成分配属性，这样就不会被保存了。这样做可以在被委托对象可能创建委托对象的情况下防止循环引用。如果委托对象保存被委托对象，那么这样的循环引用会导致任何一个对象都不会被释放。

补充一点，被委托对象被释放时，必须确保将自身作为被委托对象移除，这样委托对象就不会有一个对它的虚引用。如果没有做到这点，就会造成委托对象试图调用一个不存在的被委托对象从而产生错误。

当一个被委托对象所关注的事件发生时，`MyClass` 实例会调用合适的委托方法。该示例如

代码清单 17-8 所示。

代码清单 17-8 调用委托方法

```
@implementation MyClass
-(void)doSomethingUseful
{
    //有用的操作

    //现在我们想通知被委托对象

    [delegate requiredMethod];
}

-(void)doSomethingElse
{
    if([delegate respondsToSelector:@selector(somethingOptional)])
        [delegate somethingOptional];
}
@end
```

在想使用其中一个可选协议方法的情况下，首选需要确定被委托对象是否实现了该方法。你可以使用 `NSObject` 方法`-respondsToSelector:`，该方法会检查对象是否会响应给定方法名。这很重要。如果在调用前不检查被委托对象是否实现了可选方法并且委托没有实现该方法，就会有错误。

3. 讨论

委托模式在 Objective-C 的 Foundation 和 Cocoa 中是很普遍、很有用的模式。在框架中，在可复用组件需要应用代码的信息以处理运行时实现细节时使用。此外，开发者通常在需要将控制权转交给另一个组件时经常使用该模式，这样他们可以在次组件完全了所需处理时“被回调”。

正如第 7 章提到的，协议可以用于提供一种定义经过商定的接口的方便机制。这使得它成为了委托模式中使用的一个理想工具。在创建可复用组件时，你可以为该组件定义一个委托协议。组件的用户可以根据需要的信息或者需要提供的信息选择需要实现的方法。回顾一下协议可以指定必须和可选的方法。如果一个特定的委托行为是类要求的，就可以使用委托协议上的必须方法。另外，如果一个给定行为是可选的，使用可选方法。

学习 Objective-C 委托者模式是很有用的，因为它显示了协议和动态类型的威力。被委托对象可以是实现了委托协议的任何类。这使得该模式成为 Objective-C 中特别有用的模式。此外，Objective-C 的详尽方法命名标准和命名参数使得协议定义成为了委托对象和被委托对象之间一个不言自明的文档。从文档行为的角度看这是很强大的。也正是这个原因我特意在这里重点介绍这个设计模式。

17.2.3 将变化通知给多个对象

1. 问题

你需要在状态变化时通知多个对象。

2. 解决方案

委托模式在委托对象和委托之间存在一对一关系的情况下是一个好的选择。但是如果需要通知多个观察者状态变化应该怎么办？在这些情况下，需要实现观察者模式之类的内容，而不是实现委托者模式等一对一的关系。

观察者模式定义了一个对象可以将另一个对象注册成自身观察者的模式。对象被注册成观察者以后，任何观察者关注的事件都会在其发生时发送给观察者。

在 Objective-C 中实现观察者模式是通过 `NSNotificationCenter` 类实现的。该类为观察者和事件提供了一个全局的调度系统。观察者可以向 `NSNotificationCenter` 注册以观测系统中特定的事件。被观察对象，在事件发生时，可以发布通知到 `NSNotificationCenter`。这样，这些通知的任何观察者都可以得到通知并执行合适的操作。

利用 `NSNotificationCenter` 实现观察者模式包括两个部分。

首选，被观察对象必须在任何被观察事件发生时准备发布通知到 `NSNotificationCenter`。为此，对象访问全局的 `NSNotificationCenter` 单例，使用 `-postNotificationName:object:userInfo:` 方法。该方法接收一个 `NSString` 参数用于指定所发送通知的名称，紧跟着是发送通知的对象和可选的用户信息对象。发布通知的实现如代码清单 17-9 所示。

代码清单 17-9　发布一个通知

```
#define MY_FANCY_NOTIFICATION @"MY_FANCY_NOTIFICATION"

@implementation Bar

-(void)someMethod;
{
      ...
    [[NSNotificationCenter defaultCenter]
            postNotificationName:MY_FANCY_NOTIFICATION
                          object:self
                        userInfo:nil];
}

@end
```

通常，如上述代码所示通知名被定义成一个常量 `NSString`，这样就可以利用 Xcode 的代码补充。

在观察者一侧，观察者需要向 `NSNotificationCenter` 注册成为一个观测者。注册时，要指定想要观察的通知名称，以及可选的是要观察的对象。如果给定对象发布特定的通知，观察者就会收到通知。如果观察者将 `object` 参数指定为 `nil`，它就会收到发布给定通知的所有对象的通知。代码清单 17-10 显示了一个注册给定通知的观察者。

代码清单 17-10　注册一个观察者

```
-(void)viewDidLoad;
{
    [[NSNotificationCenter defaultCenter]
               addObserver:self
```

```
                     selector:@selector(stuffChanged:)
                         name:MY_FANCY_NOTIFICATION
                       object:nil];
}
```

在添加一个观察者对象时，你必须提供一个在收到通知时被调用的选择器。在本例中，指定的选择器是`-stuffChanged:`方法。该方法必须指定接收一个参数。在调用时，传入的参数是`NSNotification`的实例。该对象包含了有关通知的信息，发布通知的对象以及发布通知时该对象指定的`userInfo`对象。代码清单 17-11 显示了`-stuffChanged:`方法的实现。

代码清单 17-11　-stuffChanged:方法的实现

```
-(void)stuffChanged:(NSNotification *)inNotification;
{
    Bar *bar = (Bar *)[inNotification object];
    [bar askSomeQuestion];

    NSString *someData = [[inNotification userInfo] objectForKey:@"somedata"];
    [self doSomething];
}
```

最后，作为观察者的重要部分就是要确保在释放时移除观察者。如果没有这样做会导致错误，因为对象已经无效了但`NSNotificationCenter`却仍然持有对它的引用。为了从`NSNotificationCenter`移除观察者，你可以调用对象方法`-removeObserver:`。

代码清单 17-12 显示了`dealloc`方法的一个实现，其中该对象从`NSNotificationCenter`移除了观察者。



说明

对于任何观察者只需要调用一次该方法。即使你观测从不同的被观测对象接收到的多个通知，通过该方法移除观察者会彻底清除对所有通知的观测。

代码清单 17-12　移除观察者

```
@implementation Foo

-(void)dealloc;
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];
    [super dealloc];
}

@end
```

3. 讨论

`NSNotificationCenter`提供了 Objective-C 语言的观察者模式的实现。它支持对象发布它

们状态变化的通知，并使得任何想得到这些变化的通知的对象都可以观测到通知。NSNotificationCenter 支持多个对象观测通知，同时也允许多个对象发布给定通知。通过这种方式，它提供了观察者和通知者之间多对多的关系。你可能会问什么时候使用 NSNotificationCenter，什么时候使用委托模式。首先，如果有多方关注你要发布的消息，此时都必须使用 NSNotificationCenter。委托模式只能在一对一的关系中使用。你可能在使用委托模式时会试图通过一个委托数组来克服这个限制。不要这样做而应该考虑使用 NSNotificationCenter。

使用 NSNotificationCenter 比使用委托者更合理的一种情形就是在观察者和被观察对象在代码中距离比较远的时候。换句话说，如果观察者和所观测的对象在完全不同的子系统中，从应用的另一侧得到对方的引用会很困难，作为一个全局单例，NSNotificationCenter 提供了这两个对象之间简便的接口。使用 NSNotificationCenter 也有一些限制，它被设计成仅用来传递简单的数据。它无法使用一些像委托协议那样可以让委托的使用变得很方便的复杂定义。通知过程中传递的数据也仅限于对象和用户信息的字典。这并不意味无法传递复杂的数据，只是不如委托协议方便。



说明

通知相比委托者的另一个大的限制就是，委托者可以有一个返回值，而通知不行。

作为观察者模式及其 Objective-C 的实现，NSNotificationCenter 提供了一个 Objective-C 中有趣的设计模式，原因就是它是通过框架类，而不是通过 Objective-C 语言本身实现的。换句话说，仅仅通过一个标准类 NSNotificationCenter 以及观测对象和被观测对象之间协商好的标准就可以实现该设计模式，这使得它成为一个可以去研究的有趣模式。当你在自己的代码中发现设计模式，你可以利用该模式以及它是如何实现的知识来自己创建复用性更佳的组件，之后可以将组件轻松集成到代码中。

17.3　小结

本章展示了如何在 Objective-C 中实现 3 种不同的设计模式。这当然不是可用的设计模式的完整列表。本章的目标是介绍一些有代表性的设计模式，让大家了解和 C++或 Java 相比，Objective-C 这种语言如何影响设计模式的实现。我建议为了进一步研究这一主题，可以选择一本关于设计模式的书。这种书并不少，并且其中至少有一本专门介绍 Cocoa 中的设计模式。不论你如何看待设计模式，我都建议你需要努力学习它们。

第 18 章

利用 NSCoder 读写数据

18

本章概要

- ❑ 学习序列化
- ❑ 实现 NSCoding 协议
- ❑ 使用 NSArchiver 和 NSUnarchiver 来将对象归档到硬盘

很多现代语言都可以将对象编码成可以归档到硬盘或者通过网络连接传输的数据。这个过程就是序列化。这个概念就是在通过硬盘或者网络将对象从一个进程发送到另一个进程时，你需要一种将对象就地“冻结”的机制，以一种与平台无关的格式包含所有的数据，这样对象的接收者就可以在解冻并重组对象的同时保持所有数据的完整。

Objective-C 用一套类和协议来实现序列化。Objective-C 序列化的核心就是 NSArchiver 和 NSUnarchiver 类。通过它们，就可以传入符合 NSCoding 协议的对象，这样它们接收对象并将这些对象序列化成一种可以传输到硬盘或者通过网络传输的数据格式。

要使用 Objective-C 序列化，要做的第一件事就是在对象中实现 NSCoding 协议。

在对象上实现NSCoding协议

NSCoding 协议定义了两个为了符合 NSCoding 协议所必须实现的方法。第一个就是-encodeWithCoder:方法。该方法会在需要序列化对象时被归档程序调用。它接收一个 NSCoder 作为参数。NSCoder 是一个抽象基类。通常传递给你的实际对象将是 NSArchiver 或者 NSKeyedArchiver 的实例。使用 NSCoder，你要将对象的成员变量归档到其中。

对象编码

为了将成员变量序列化到 NSCoder 中,要使用传入到-encodeWithCoder:方法的 NSCoder 实例上的方法。NSCoder 还提供了很多为基本数据类型编码以及对象编码的不同方法。为了将一个对象编码到 NSCoder，你需要使用 NSCoder 方法-encodeObject:或者-encodeObject:forKey:。不是所有的 NSCoder 实例的工作原理都一样。一些支持键控归档，一些则不支持。为了确认 NSCoder 的一个实例是否支持键控归档，可以使用-allowsKeyedCoding 方法。如果支

持键控归档，该方法就返回真。任何支持 `NSCoding` 协议的对象都可以通过`-encodeObject...`方法进行编码。`NSString`、`NSArray` 和 `NSDictionary` 等大多数底层的 Foundation 类都支持。使用`-encodeWithCoder:`方法进行对象编码的示例如代码清单 18-1 所示。

代码清单 18-1　简单的`-encodeWithCoder:`实现

```
@interface MyClass : NSObject <NSCoding>
{
    Foo *memberVariable;
    Bar *anotherVariable;
    NSArray *someMemberArray;
}

@end

@implementation MyClass

...

-(void)encodeWithCoder:(NSCoder *)inCoder
{
    if([inCoder allowsKeyedCoding])
    {
        [inCoder encodeObject:memberVariable forKey:@"memberVariable"];
        [inCoder encodeObject:anotherVariable forKey:@"anotherVariable"];
        [inCoder encodeObject:someMemberArray forKey:@"someMemberArray"];
    }
    else
    {
        [inCoder encodeObject:memberVariable];
        [inCoder encodeObject:anotherVariable];
        [inCoder encodeObject:someMemberArray];
    }
}

@end
```

如果编码器不支持键控编码，所有的变量都按照传入 `NSCoder` 的顺序进行编码。因此，按相同的顺序进行解码至关重要，解码的代码必须也知道顺序。基于此原因，优先使用键控归档程序而不是非键控归档程序。

键控归档被视为更“现代”的归档形式。如果不需要向后兼容非键控归档程序，你可以安全地实现仅使用键编码的代码。



说明

确保在接口声明中声明实现了 `NSCoding` 协议。

基本类型编码

整型、浮点型和结构体等基本标量也可以通过-encodeDouble:forKey:, -encodeInt:forKey:等在 NSCoder 上定义的方法进行编码。如果扩展上个示例中的类以包含一些标量和结构体，可以按代码清单 18-2 所示修改-encodeWithCoder:方法。

代码清单 18-2　-encodeWithCoder:方法

```
@interface MyClass : NSObject <NSCoding>
{
    Foo *memberVariable;
    Bar *anotherVariable;
    NSArray *someMemberArray;
    NSRect aRect;
    int aNumber;
}

@end

@implementation MyClass

...

-(void)encodeWithCoder:(NSCoder *)inCoder
{
    if([inCoder allowsKeyedCoding])
    {
        [inCoder encodeObject:memberVariable forKey:@"memberVariable"];
        [inCoder encodeObject:anotherVariable forKey:@"anotherVariable"];
        [inCoder encodeObject:someMemberArray forKey:@"someMemberArray"];
        [inCoder encodeRect:aRect forKey:@"aRect"];
        [inCoder encodeInt:aNumber forKey:@"aNumber"]
    }
    else
    {
        [inCoder encodeObject:memberVariable];
        [inCoder encodeObject:anotherVariable];
        [inCoder encodeObject:someMemberArray];
        [inCoder encodeRect:aRect];
        [inCoder encodeInt:aNumber];
    }
}

@end
```

NSCoder 有很多不同的方法可用。更多信息参见 NSCoder 文档。

使用对象图

考虑到 NSCoding 协议的工作方式，给定对象需要对成员变量编码，其中每个成员变量还需要对其成员变量编码，以此类推。NSCoder 的一个好的方面就是仅需要关注自身状态的编码。

假设所有的成员变量都实现了 NSCoding 协议，你就可以“不透明”对它们进行编码而且不需要考虑它们内部有什么。

这就是说，你还可能需要关注循环引用。NSCoder 不会进行任何类型的循环引用检查，如果在代码中有循环引用，就会导致归档问题。程序员可能没有意识到的另一个陷阱就是可以被归档的只有数据本身，而不是对象的指针或者地址。这就意味着归档一个对象指针，解档后就会和原来的指针不一样。因此，在解档时需要手动地重新连接引用，这样才真正完成了对象图。

使用其他类型的数据

对于一些不是标量类型或者无法成为对象的数据，你可能需要利用 NSData 装箱来对数据进行编码。为此，仅需要将数据装箱到一个 NSData 对象，然后通过-encodeDataObject:这一 NSCoder 方法对 NSData 对象编码。其中的一个示例如代码清单 18-3 所示。

代码清单 18-3　在 NSCoder 中编码一个经过 malloc 的内存块

```
@interface MyClass : NSObject <NSCoding>
{
    Foo *memberVariable;
    Bar *anotherVariable;
    NSArray *someMemberArray;
    NSRect aRect;
    int aNumber;
    char *buf;
}

@end

@implementation MyClass

//buf 可能通过 malloc(1024) 等指令分配

...

-(void)encodeWithCoder:(NSCoder *)inCoder
{
    if([inCoder allowsKeyedCoding])
    {
        [inCoder encodeObject:memberVariable forKey:@"memberVariable"];
        [inCoder encodeObject:anotherVariable forKey:@"anotherVariable"];
        [inCoder encodeObject:someMemberArray forKey:@"someMemberArray"];
        [inCoder encodeRect:aRect forKey:@"aRect"];
        [inCoder encodeInt:aNumber forKey:@"aNumber"]
        NSData *bufData = [NSData dataWithBytes:buf length:1024];
        [inCoder encodeObject:bufData forKey:@"someData"];
    }
    else
    {
        [inCoder encodeObject:memberVariable];
        [inCoder encodeObject:anotherVariable];
```

```
        [inCoder encodeObject:someMemberArray];
        [inCoder encodeRect:aRect];
        [inCoder encodeInt:aNumber];
        NSData *bufData = [NSData dataWithBytes:buf length:1024];
        [inCoder encodeDataObject:bufData];
    }
}

@end
```

解码对象

NSCoding 协议的另一方面就是用来将编码的对象解码。有一个名为-initWithCoder:的特殊初始化函数专门用于此操作，这是唯一一个在使用时不需要调用指定初始化函数的初始化函数。使用该方法进行解码的实现示例如代码清单 18-4 所示。

代码清单 18-4　-initWithCoder:示例

```
-(id)initWithCoder:(NSCoder *)inCoder
{
    if((self = [super init]))
    {
        if([inCoder allowsKeyedCoding])
        {
            memberVariable =
                    [[inCoder decodeObjectForKey:@"memberVariable"]
                              retain];
            anotherVariable =
                    [[inCoder decodeObjectForKey:@"anotherVariable"]
                              retain];
            someMemberArray =
                    [[inCoder decodeObjectForKey:@"someMemberArray"]
                              retain];
            memberVariable =
                    [[inCoder decodeObjectForKey:@"memberVariable"]
                              retain];
            aRect = [inCoder decodeRectForKey:@"aRect"];
            aNumber = [inCoder decodeIntForKey:@"aNumber"];
            NSData *bufData = [inCoder decodeObjectForKey:@"someData"];
            buf = malloc(1024);
            [bufData getBytes:buf length:1024];
        }
        else
        {
            memberVariable = [[inCoder decodeObject] retain];
            anotherVariable = [[inCoder decodeObject] retain];
            someMemberArray = [[inCoder decodeObject] retain];
            memberVariable = [[inCoder decodeObject] retain];

            aRect = [inCoder decodeRectForKey:@"aRect"];
            aNumber = [inCoder decodeIntForKey:@"aNumber"];
```

```
            NSData *bufData = [inCoder decodeDataObject];
            buf = malloc(1024);
            [bufData getBytes:buf length:1024];
        }
    }
    return self;
}
```



警告

-initWithCoder:被指定为 NSCoding 协议的一部分。如果所继承的对象实现了 NSCoding 协议，你必须在这里调用[super initWithCoder:...]，而不是[super init]。在本例中，类从没有实现该协议的 NSObject 继承。对于-encodeWithCoder 方法也是如此。如果父类实现了 NSCoding 协议，它也必须调用父类的-encodeWithCoder 方法。

特别提示，应该确保保留了从 NSCoder 获得的对象。它们遵循在其他地方使用的相同的保留/释放规则。但是标量和结构体不需要被保留，因为它们不是对象。

18.1　使用 NSArchiver 和 NSUnarchiver

如果对象支持 NSCoding 协议，就可以使用 NSCoder 对它们进行归档和解档。最常用的 NSCoder 就是 NSArchiver 和 NSKeyedArchiver 以及它们相应的解码类 NSUnarchiver 和 NSKeyedUnarchiver。再次说明，NSArchiver 被视为一个遗留类，所以这里我会介绍 NSKeyedArchiver。

通过 NSKeyedArchiver 进行一个对象图归档，可以使用类方法+archivedData-WithRootObject:，该方法会返回一个 NSData 数据，包含了根对象以下的所有归档数据。此外，还可以使用工厂方法+archiveRootObject:toFile:直接将数据写入文件。示例如代码清单 18-5 所示。

代码清单 18-5　将数据写入硬盘

```
-(void)writeDataToPath:(NSString *)inPath
{
    [NSKeyedArchiver archiveRootObject:objects toFile:inPath];
}
```

通过该调用,NSKeyedArchiver 会从所提供的根对象开始,通过调用-encodeWithCoder:将自身作为编码者，然后从中获取数据并写入硬盘。由于-encodeWithCoder:会调用根对象的所有子对象的-encodeWithCoder:，它们就会被编码到文件中。从一个存档文件读取数据也一样简单。为此，只需要使用 NSKeyedUnarchiver 的+unarchiveObjectWithData 方法

来解码一个 NSData 对象，或使用+unarchiveObjectWithFile:来解码文件。示例如代码清单 18-6 所示。

代码清单 18-6　从硬盘读取数据

```
-(void)readDataFromPath:(NSString *)inPath
{
    objects = [[NSKeyedUnarchiver
                  unarchiveObjectWithFile:inPath] retain];
}
```

再次指出,NSKeyedUnarchiver 会打开文件并调用文件中根对象的 initWithCoder 方法。这会使得所有对象图中的子对象都可以重新生成。

18.2　处理存档文件格式和遗留数据

利用 NSKeyedArchiver 编写的归档文件可以是 XML 或是二进制的。为了配置文件格式，可以在将数据写入硬盘之前调用 NSKeyedArchiver 对象的-setOutputFormat:方法。（这就要求在创建归档程序时使用标准的 alloc/init 方法）。你可以将该值设成 XML 格式的 NSPropertyListXMLFormat_v1_0 或者二进制格式的 NSPropertyListBinaryFormat_v1_0。二进制格式往往比 XML 速度稍快并且小些，移植性当然也会差些。

NSCoder 系统是为主要由 Objective-C 对象构成的现代对象图设计的。如果你有遗留的二进制文件格式，总是可以转而使用标准的 C 文件 I/O 例程进行读写。

18.3　小结

Objective-C 的对象序列化容易使用并且很强大。你可以使用它来将应用状态存储到硬盘或者通过网络连接将数据从一个进程发送到另一个进程。本章展示了如何让对象遵守 NSCoding 协议，以及如何通过 NSKeyedArchiver 和 NSKeyedUnarchiver 将对象图读出和写入硬盘。总之，这会给你使用 Objective-C 的序列化所需的工具。

第 19 章

在其他平台上使用 Objective-C

19

本章概要

- 在 Windows、Linux 和其他平台使用 Objective-C
- 深刻理解 Objective-C 框架的成熟度
- 使用其他类库

虽然到目前为止最好的 Objective-C 编码平台来自苹果公司，但它们绝不仅适用于苹果公司的平台。Objective-C 在 Linux、BSD 甚至 Windows 等其他平台都有相当久远的历史。根据具体需求，你会发现一些能很好地支持这些替代平台的开源社区。本章将简要介绍一些其他的平台，并告诉你在哪里可以找到更多关于它们的信息。

在其他平台上使用 Objective-C 时面临的最大的挑战在于对能使 Objective-C 变得强大的框架的支持。移植 Objective-C 语言是一件琐碎的事。由于 GNU 编译器集合（gcc）开始支持 Objective-C，Objective-C 在几乎所有 gcc 支持的平台都可用。但是，移植核心框架是一项更加艰巨的任务。

可以确定的是，Foundation 框架已经有最广泛的跨平台支持。由于本书几乎只用到了 Foundation 框架，这意味着除了小部分特殊情况外，本书的所有例子在其他平台都可以编译运行。

而 Cocoa 和一些高层框架通常在其他平台上就不可用了。也有例外情况，不过老实说，如果你要在 OS X 之外的平台运行应用，在考虑使用 GUI 框架时就要特别小心。

也就是说，能最好地支持通常和 Objective-C 一起使用的所有框架的主要有两个项目。它们是 GNUstep 和 Cocotron。这两个开源项目在移植性技术方面使用完全不同的方法，不过结果是一样的，即支持在 Linux、Windows、BSD 和其他平台编写和编译使用 Cocoa 和 Foundation 的 Objective-C 代码。

19.1 使用 GNUstep

这些项目中最古老的项目是 GUNstep 项目，它的历史可以一直追溯到最初的 NeXTstep 时代。事实上，这个项目最初的开发是为了提供一个闭源项目 NeXTstep 平台的开源替代品。为此，它

很好地重建了 NeXTstep 的桌面环境，包括图标、文件浏览器、邮件客户端等。参见图 19-1。

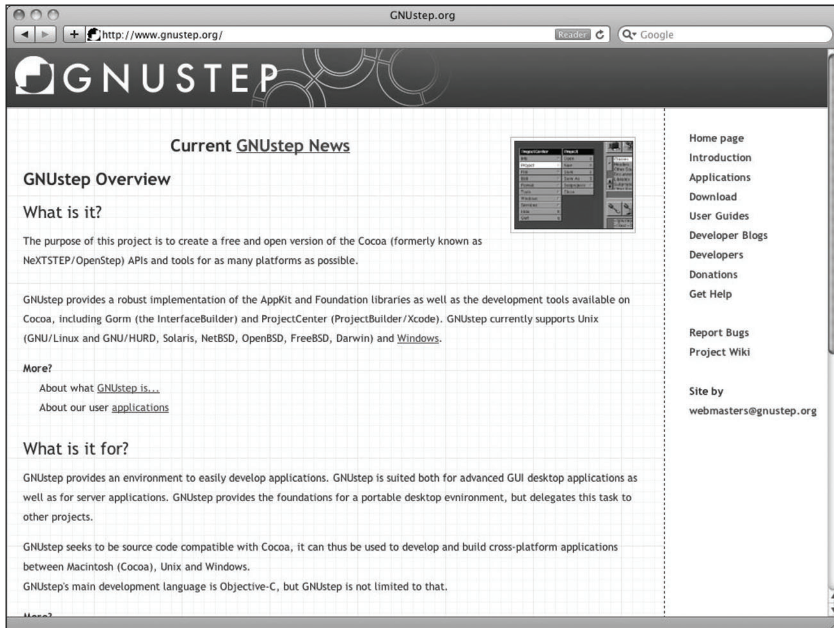


图 19-1　运行在 Unix 上的 GNUstep 环境

因为其久远的历史，GNUstep 项目对 Foundation 和 Cocoa 有一些最好的支持。不过，因为它们的真正的目标是复制 NeXTstep 环境，而不是模拟运行应用的 Mac OS X 或平台的原生组件集，而它们实际上包含了整个 NeXTstep 环境的组件集。这意味着如果你选择通过该项目将应用移植到 Windows，你的应用在 Windows 上运行时，看起来像一个 NeXTstep 应用。这包括所使用的菜单类型的方方面面。此外，还有一些和运行 GNUstep 应用所需的文件系统相关的问题。最后要确认的一点，为了在 Windows 上运行一个 GNUstep 程序，你必须安装完整的 GNUstep 文件系统以及众多支持库。

这一切可以让一个普通用户困惑。出于完整性考虑，在这里需要提到的是，过去几年间为 GNUstep 添加皮肤支持做了一些努力。这样你至少可以创建一个 Windows 应用，并使用 Windows 的组件集。换言之，GNUstep 致力于图形应用的可移植性，并不断改善，但是目前还说不上完美。总之，对于 Foundation 框架，GNUstep 的项目有一些到处可用的最好、最全面的支持。我想说如果你的应用是一个命令行应用，比如服务器，并且打算移植到 Windows 或 Linux，那么 GNUstep 项目将很可能可以帮助你实现这些。可以说，利用 Mac OS X 上已经写好的现成代码并移植到 Windows 的一个较好的方式就是利用 Objective-C 和 GNUstep 移植后台、底层非界面的代码。你可写一个原生的图形应用，通过进程间通信进行通信或者将它作为库来链接。这样做可以两全其美。一方面统一了业务逻辑代码，同时还能向用户提供熟悉的原生用户界面。

关于 GNUstep 项目的更多信息，可以访问它们的站点 http://www.gnustep.org。

19.1.1 使用Cocotron

另一个最近的跨平台Objective-C方面的尝试就是Cocotron项目的创建。在将应用移植到Mac OS X之外的平台方面，Cocotron项目采用不同的方式。Cocotron为Xcode提供了一个的交叉编译器环境，这样你可以在Mac OS X平台上Xcode里交叉编译你的应用。通过该交叉编译器，你可以编译Windows、Linux或其他UNIX桌面版本。以这种方式交叉编译的应用，观感和行为都和原生的应用一致。

Cocotron的工作原理就是利用Xcode在编译代码时支持多个工具链和SDK的能力。为iPhone OS或者Mac OS X编译代码时，只需要单击一个按钮的技术使得Cocotron可以发挥它的威力。

如果你觉得将Mac OS X和Xcode作为开发环境最惬意，那Cocotron是一个完美的解决方案。它支持在Mac OS X上进行所有开发，然后简单地改变SDK并为目标平台重新编译应用。

这似乎还不够，Cocotron还有一些对Cocoa和Foundation框架的最好的第三方支持。它的实现已足以开发同时部署在Mac OS X和Windows平台上的Cocotron商业级应用。

Cocotron相对落后的最大领域在网络、线程处理和一些高层次的框架支持。不过正在积极努力中，而且如果你有足够的预算，你甚至可以和 Cocotron 的维护者订立契约来改善你的应用所需要的具体的库。

图19-2展示了一个使用Cocotron编写的运行在Windows和Mac OS X上的应用。

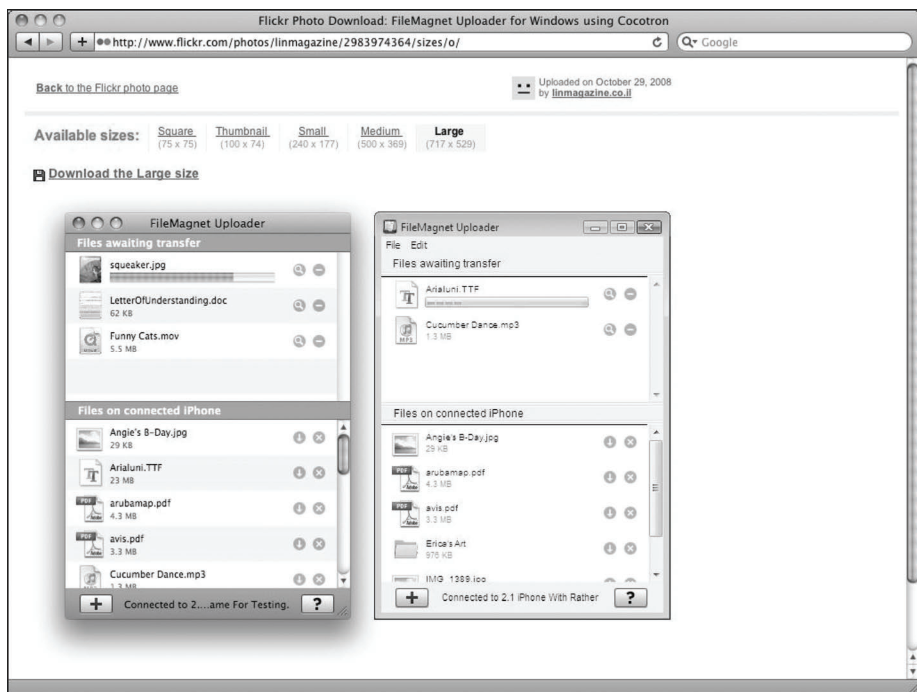


图19-2 使用Cocotron编写的应用

如果你不方便将 Mac OS X 作为主要的开发环境，Cocotron 可能不适合你。不过，如果你的目标是通过尽可能小的改动将 Mac OS X 上的应用移植到 Windows 上，那 Cocotron 应该是一个好选择。最起码我会说，下载项目并尝试用它编译你的应用。用不到一天的时间，你就会知道你的应用是否是那些可以简单通过这个项目来移植的应用。

关于 Cocotron 项目的更多信息，可以访问它的站点 www.cocotron.org。

19.1.2　使用其他开源库

Cocotron 和 GNUstep 不是仅有的苹果 Objective-C 框架的开源替代方案。还有一些其他的实现，不过它们大多数局限于一定领域或不再维护了。我的建议是，考虑在 Mac OS X 或 iPhone 之外的平台上使用 Objective-C 时，评估一下我这里提到的两个项目，看它们是否满足你的需求，如果是，那你很幸运!

19.2　展望未来

我想写本书的一个原因是我觉得 Objective-C 是一个未得到充分重视的语言。它的“姐妹”C++、Java 和其他语言都受到更多的关注。我认为其中的一个原因是 Objective-C 通常只在 Mac OS X 上进行开发时用到。

随着 iPhone 和 iPad 的出现，Objective-C 得到了大量新的关注，而且学习语言的新用户出现了前所未有的增长。这有利有弊。

好的一面是因为这可以确保这个平台还会存活许多年，但不好的一面是它显露了这个优秀的语言在标准化和文档化方面的不足。

不像 C 和 C++，Objective-C 没有一个 ISO 标准组织来推动它的规范。有些人可能会说这其实是一件好事。事实上，这也许是 Objective-C 优雅而且非常适合它的任务的原因之一。然而，问题是由于缺少标准化，Objective-C 在它的核心平台之外很少使用。这意味着以后这个平台上新的开发者都被限制在 Mac OS X 或者 iPhone OS 上进行开发。

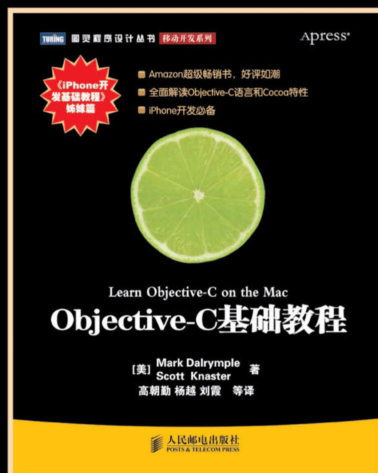
没有人知道未来可能有哪些新的平台和硬件。学习一门语言，就是对这门语言和它运行所依赖的平台上进行智力投资。为了确保在 Objective-C 上的投资可以在未来获取收益，我们需要在核心平台之外推动 Objective-C 语言的发展。这需要得到 UNIX 和 Windows 等其他平台的更多的支持和采用。我不敢说支持 Objective-C 的 ISO 标准化，但是我肯定乐意看到第三方项目在跨平台支持方面的改进。

可以肯定的是本章向你介绍的一些项目就是以此为目标，这是非常好的，我们应该帮助并推动这些项目发展，尽管它们可能并不是在我们自己喜欢的平台上。让 Objective-C 存在下去并有一个支持跨平台的编程环境，有助于你在这里的投资在未来更有价值。

19.3 小结

本章介绍了若干开源项目，它们支持你在 Mac OS X 和 iPhone OS 之外的平台编译和运行 Objective-C 应用。虽然它们还不够完美，但都很不错，而且一直在完善，以后它们可能提供一个可用的开发平台。时至今日，有一些开发者正在使用这些第三方项目来开发商业级应用。这些开发者在使用这些第三方项目进行移植遇到困境时，他们也可以和这些项目的维护者一起完善这些项目。如果可能，我鼓励你这样做。这不仅有助于完善这些替代平台的语言支持，同时对提高你对语言的理解也有帮助。最好的学习环境之一是使用现有代码并尝试完善它。

促进 Objective-C 的替代平台生态圈的繁荣，有助于确保我们的智力投资在将来相当长的一段时间会一直有回报。


图灵程序设计丛书 移动开发系列
Apress
《iPhone开发基础教程》姊妹篇
Amazon超级畅销书，好评如潮
全面解读Objective-C语言和Cocoa特性
iPhone开发必备
Learn Objective-C on the Mac
Objective-C基础教程
[美] Mark Dalrymple Scott Knaster 著
高朝勤 杨越 刘霞 等译
人民邮电出版社
POSTS & TELECOM PRESS


图灵程序设计丛书 移动开发系列
Apress
解析iOS设计模式的开山之作
优化Objective-C编程实践的必修宝典
由此迈入移动开发高手行列
Pro Objective-C Design Patterns for iOS
Objective-C编程之道
iOS设计模式解析
[美] Carlo Chung 著
刘 威 译
人民邮电出版社
POSTS & TELECOM PRESS


TURING
cocos2d-x
手机游戏开发
跨 iOS、Android 和沃 Phone 平台
徐松林 黄 猛 编著
人民邮电出版社
POSTS & TELECOM PRESS

好学的Objective-C

Objective-C Developer Reference

要为市面上最热门的Mac、iPhone和iPad等设备创建应用，就必须掌握Objective-C和面向对象编程。本书作者是顶尖的Mac开发人员和专业作家。通过本书的详尽指引，即使是编程新手也可以迅速学会Objective-C。本书全方位地介绍了Objective-C，从基础知识到资深程序员所使用的高级技术等众多主题，其中包括内存管理、多个框架的结合使用、线程安全的技巧，以及Xcode的详细用法等。

通过阅读本书，读者将能够：

- 掌握Objective-C语法、运行时和Xcode，编写出第一个移动应用程序
- 创建类，使用属性，了解对象
- 使用代码块、线程、KVO和协议
- 定义和编写宏，处理错误并在项目中使用框架
- 清理线程，学会使用设计模式，掌握高级技术
- 利用NSCoder读写数据
- 为Windows、Linux和其他平台编写代码

本书既能引导Mac、iPhone和iPad开发新手入门，又可帮助高级程序员提高技能，是Objective-C开发人员的案头必备书籍。

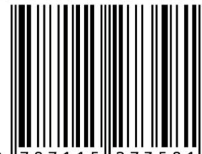


www.wiley.com

图灵社区：www.ituring.com.cn
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计/移动开发

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-27358-1
定价：55.00元